UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

TESIS DOCTORAL

De programas abstractos a cotas asintóticas precisas en forma cerrada

From Abstract Programs to Precise Asymptotic Closed-Form Bounds

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Diego Esteban Alonso Blas

Directores

Purificación Arenas Sánchez

Samir Genaim

Madrid, 2014

De Programas Abstractos a Cotas Asintóticas Precisas en Forma Cerrada

TESIS DOCTORAL

Memoria de Tesis Doctoral presentada para obtener
el grado de *Doctor en Ingeniería Informática* por
Diego Esteban Alonso Blas

bajo la dirección de los Profesores
Purificación Arenas Sánchez
Samir Genaim

Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

Abril de 2014

From Abstract Programs to Precise Asymptotic Closed-Form Bounds

PH.D. THESIS DISSERTATION

A PH.D. Thesis Dissertation, presented to obtain
the degree of PH.D. *in Computer Science* by

Diego Esteban Alonso Blas

with the advice of Professors
Purificación Arenas Sánchez
Samir Genaim

Department of Computational Systems
School of Computer Science
Complutense University of Madrid

April 2014

Preface

This document is a Dissertation of a PH.D. Thesis, for obtaining the degree of PH.D. *in Computer Science*, issued by the Complutense University of Madrid (UCM), with the *Special Mention of European Ph.D.*

Supporting Publications. A PH.D. Thesis is backed by a set of **Supporting Publications**, which are scientific articles coauthored by the PH.D. candidate and published in journals or in the proceedings of international conferences of adequate prestige. This PH.D. Thesis is supported by the following articles. For each article, we show such quality indicators as the number of citations [1], and the average acceptance rate and the CORE ranking[2] of the conference.

ELVIRA ALBERT, DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, SAMIR GENAIM, AND GERMAN PUEBLA. Asymptotic Resource Usage Bounds. In Zhenjiang Hu, editor of the Proceedings of *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 5904 of *Lecture Notes in Computer Science*, pages 294–310. Springer, December 2009. CORE: B. Acceptance rate: 37%. Citations: 6.

DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Handling Non-linear Operations in the Value Analysis of COSTA. *Electronic Notes in Theoretical Computer Science*, 279(1):3–17, 2011. Proceedings of Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE). Citations: 1.

DIEGO ESTEBAN ALONSO-BLAS AND SAMIR GENAIM. On the Limits of the Classical Approach to Cost Analysis. In Antoine Miné and David Schmidt, editors of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 7460 of *Lecture Notes in Computer Science*, pages 405–421. Springer, September 2012. CORE: A. Acceptance rate: 38%. Citations: 7.

DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Precise Cost Analysis via Local Reasoning. In Dang Van Hung and Mizuhito Ogawa, editors of the Proceedings of *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 8172 of *Lecture Notes in Computer Science*, pages 319–333. Springer, October 2013. CORE A. Acceptance rate: 33%.

[1] As recorded by the Google Scholar web site, `scholar.google.com`

[2] Ranking of conferences in Computer Science published in 2013 by the *Computer Research and Education Association (CORE)*, `www.core.edu.au`

Structure of the Dissertation. This PH.D. Dissertation is structured as a *Dissertation by Publications* with *Special Mention of European* PH.D.. It contains a compilation of the Supporting Publications, accompanied by a Summary of the Research that includes

This Dissertation is split in the following parts.

- Part I contains a Summary of the Research, that includes an overview of the state of the art on the subject, the goals and hypothesis, a synthetic discussion, and the conclussions. This text is written in British English.
- Part II contains, for the convenience of Spanish readers, a translation of Part I to the Castilian language.
- A bibliography in English figures after Part II.
- Part III attaches the Supporting publications of this Dissertation. For each article we show, framed and scaled, the pages as they appear in the corresponding proceedings or journal, as available in the digital libraries of the publishers Springer and Elsevier. We also show, unframed, the appendices for each publication.

This Dissertation was edited and compiled with LaTeX. The Utopia serif typeface was used to format the text of the Dissertation, and `Inconsolata` was used as the monospace font in the code listings.

Acknowledgements

This Dissertation has been made possible and successful thanks to the help and support of a lot of people, to whom I would like to express my gratitude.

First, I thank my PH.D. advisors Puri and Samir, for their advice. Without it, it would not have been possible to complete my PH.D.. I thank them for introducing me to the world of scientific research, and for describing me with a deep honesty the methods, practises, and compromises, of scientific writing and publishing. I am grateful for having benefitted from their expertise in writing in this field, expertise that allowed us to improve the quality of the articles. I also thank them by granting me their time whenever I felt that I needed it. In particular, Samir he deserves a special mention for revealing me how relevant our contribution in our 2012 article was, for this article came at a crucial moment and helped to ensure the viability of my PH.D.. I also thank Puri for her expertise on the fine points of mathematical writing, for her thoroughness and I also thank her for counseling me on how to deal with the red tape.

As no research is done in the void, I must acknowledge those whose work made it possible to start mine. In this sense, I've had the luck of being associated with the group of researches that have developed the COSTA system, which proveded me with an useful environment for developing the contributions of this Thesis. I thank them for their help in whatever problem I had with the installation and configuration of this system. I also owe a special recognision to Elvira, with whom a chance encounter six and a half years ago initiated me in the subject that would become the subject of my PH.D. and of this dissertation.

An important part of my experience was my research stay for three months in the city of Münich. Thus, I am very grateful to Steffen Jost for offering me the opportunity to work with him, as well as to the people with whom I socialised during my stay.

I also want to mention my experience as scholar and teacher in the Faculty, and the deep and interesting conversations on several subjects, which exposed me to the particular world of Academia, with its lights and shadows.

Finally, I thank the members of my family, in so far as they encouraged me to initiate, continue, and terminate my PH.D.

Institutional and Financial Support

Positions. The major support for my PH.D. was provided by the Complutense University of Madrid. In 2009 I was supported by a contract as affiliated researcher, from the Research Project MERIT-FORMS-UCM. Also, from 2010 to 2013 I was supported by a four-years PH.D. scholarship *(Beca Complutense Predoctoral)*, granted by the UCM in 2009.

Research Projects. During the realisation of my PH.D., I have been affiliated to the following Research Projects:

- DOVES (Development Of Verifiable and Efficient Software), a project financed by the Government of Spain, TIN-2008-05624.
- PROMETIDOS *(PROgrama de MÉTodos rIgurosos de DesarrollO de Software)*, S2009TIC-1465, financed by the Regional Government of Madrid.
- GPD *(Grupo de Programación Declarativa)*, a research group in the Complutense University of Madrd, UCM-BSCH-GR35/10-A-910502.

These research projects financed my assistance to the conferences in which the supporting publications were presented.

Abstract

Cost analysis (a.k.a. resource usage analysis) aims at *statically* determining the amount of resources required to safely execute a given program, i.e., without running out of resources. By *resource*, we mean any quantitative aspect of the program, such as memory consumption, execution steps, etc. Several cost analysis frameworks are available, although different in their underlying theory, all of them usually report the cost of a program as a closed-form upper bound function. Inference of lower bound functions has also been considered. However, it has received less attention.

The important properties of a cost analyser are applicability, precision, and scalability. These properties are often antagonistic, e.g., achieving good scalability is typically done by sacrificing precision, and thus, a cost analyser might excel in one property, and fall behind in another. In this thesis, we aim at exploring the gap between the different existing approaches to cost analysis, to develop techniques that are able to infer *asymptotically* precise bounds, and, at the same time, have good scalability and applicability properties.

Our point of departure is the *classical approach* to cost analysis, which is widely applicable, scalable, and has a reasonable precision in practice. However, for some programming patterns it infers bounds which are asymptotically less precise than other approaches. The workflow of this approach is divided into several phases: (I) transforming the input program into an abstract program by replacing data structures by some notion of *size*; (II) transforming the abstract program to a set of recursive equations that describes the program's cost in terms of the input data; and (III) solving these equations into closed-form bounds.

Our contributions are related to steps II and III above, and consist of analyses, transformations, and solving techniques that allow going ***From Abstract Programs to Precise Asymptotic Closed-Form Bounds***. We argue that they close the precision gap with respect to other approaches, in particular the amortised analysis approach, while thet still exhibit good scalability and applicability properties. In addition, some of the contributions bring a methodological innovation in this thesis, which is to apply *The Calculus of Computation* to cost analysis.

Keywords: Static Analysis. Cost Analysis. Automatic Complexity Analysis. Closed-Form Bounds.

Resumen

En el análisis de coste (o de consumo de recursos) se estudia cómo determinar *estáticamente* cuántos recursos se requiere para ejecutar un programa, entendiendo por *recurso* cualquier medida cuantitativa del programa, tales como el consumo de memoria, el número de pasos de ejecución, etc. Existen varios métodos de análisis de coste que, aunque se basan en teorías diferentes, normalmente dan el coste de un programa como una función de cota superior en forma cerrada. También se ha estudiado, aunque menos, cómo inferir funciones de cota inferior.

Las propiedades importantes de un analizador de coste son su aplicabilidad, su precisión, y su escalabilidad. Éstas a veces son antagónicas, por lo que para lograr buena escalabilidad se suele sacrificar la precisión, y por ello un analizador de coste puede destacar en una propiedad pero no en la otra. En esta tesis exploramos la distancia entre los enfoques existentes de análisis de coste, para desarrollar técnicas que puedan inferir cotas *asintóticamente* precisas, a la vez que tengan una buena escalabilidad y aplicabilidad.

Nuestro punto de partida es el *enfoque clásico* al análisis de coste, el cual es ampliamente aplicable, escalable, y en la práctica tiene una buena precisión. Sin embargo, para algunos programas infiere cotas que son asintóticamente menos precisas que las que infieren otros enfoques. En este enfoque, el flujo de trabajo se divide en varias fases: (I) se transforma el programa de entrada en un programa abstracto, reemplazando las estructuras de datos por una noción de *tamaño*; (II) se transforma el programa abstracto en un conjunto de ecuaciones recursivas, que describen el coste del programa en función de los datos de entrada; y (III) se resuelven estas ecuaciones a cotas en forma cerrada.

Nuestras contribuciones se hallan en los pasos II y III, y consisten en análisis, transformaciones, y técnicas de resolución que permiten ir ***De Programas Abstractos a Cotas Asintóticas Precisas en Forma Cerrada***.

Creemos que éstas resuelven la diferencia de precisión con respecto a otros enfoques, en particular con el de análisis amortizado, manteniendo una buena aplicabilidad y escalabilidad. Además, nuestras contribuciones aportan una innovación metodológica, que es aplicar el *Cálculo de la Computación* al análisis de coste.

Palabras Clave: Análisis Estático. Análisis de Coste. Análisis Automático de Complejidad. Cotas en Forma Cerrada.

Contents

Part I

Summary of Research

1 | Introduction

Every machine has a functionality that it is built to perform, and an environment with which it interacts. A resource, in this context, is any entity in the environment that can be used by the machine. To safely run a machine, we must supply its environment with all the resources it needs, and thus, it is often crucial to know the amount of such resources beforehand.

In the concrete case of a computer program, its functionality is to process some input data and generate some output. Its runtime environment includes, among others, the hardware on which it is being executed such as memory, processors, etc. A **resource** in this context can be a physical magnitude like time or energy; a logical element of hardware like a CPU cycle, a word of memory, or a network packet; it may be a logical unit of software, like an object, a method call, or a file; a resource may even be the money that a user has to pay for a billable operation, like sending a text message on a mobile phone. These resources are **consumed** in different ways: time simply passes while the program executes, memory cells are temporarily occupied, method calls are simply performed, etc.

There are two ways to estimate the **resource consumption** (or **cost**) of a program: dynamic and static. The dynamic or experimental approach uses simulators to run the program on some input data, and measures the amount of resources consumed. This approach is easy to perform, and gives exact results for the chosen input. However, a major drawback is its incompleteness since the results are valid only for the chosen input. Unlike the dynamic approach, the static or analytical approach can obtain information on the resource consumption of *all* possible executions of a program without executing it. This approach is based on the following idea:

> "Computer programming is an exact science in that [...] all the consequences of executing [a program] in any given environment can, in principle, be found out from the text of the program itself." [72]

Discovering such consequences "from the text of the program itself" without executing it, is known as a **static analysis**.

The benefit of the static approach is that, since it gives information about all possible executions, a good static analysis is by itself enough to verify that a program always works as expected. The major drawback of this approach is that it is too difficult and expensive, so manually analysing large programs is often unfeasible. A static analysis requires a mathematical expertise that only a few highly trained specialists have, and even for them a manual analysis of

a simple program is tedious, error-prone, and takes a long time. For a static program analysis to be widespread, we need to perform it automatically, by a computer program.

1 Static Cost Analysis

Automatic static analysis [100] deals with building a program, called **static analyser**, that takes as input the code of a program (in a given language) and computes some information about its runtime behaviour. This thesis lies in the field of **automatic static cost analysis**, which deals with building a computer program, called **cost analyser**, whose functionality is to take as input a program, and compute[1] the amount of resources that the program would consume when executed on some given input data. A cost analyser is usually parametric on the resource to be measured, this is done by using **cost models** which are functions that map instructions to their corresponding costs.

The output of a cost analyser is a **cost function** that maps program executions to their costs. Due to the undecidability of the underlying problems, this function typically does not describe the exact cost of the program, but it is rather an **upper bound function** (UBF for short) or a **lower bound function** (LBF for short) on the actual cost[2]. Moreover, as it is customary in the field of analysis of algorithms [45, 118], these bound functions are defined over a set of numerical input variables, and are given in **closed-form**, that is, deterministic and without recursion. To handle non-numerical input data, cost analysers use **size measures** to map such data to numerical values, e.g., length of a list.

Due to the use of size measures, a cost function is actually a UBF on the *worst-case* cost, or an LBF on the *best-case* cost. This is because two different inputs with different cost might be mapped to the same abstract value. This is also true when the programming language under consideration has some nondeterministic features, because in such case a given input might have different costs depending on some nondeterministic choice.

In the field of analysis of algorithms, a major concern is the scalability of the cost of an algorithm or a program with respect to the size of its input. To answer this kind of questions, it is common to use the asymptotic notations Big O, Big Omega and Big Theta [88]. Intuitively, these notations indicate that, for big values of the input, one function is smaller, greater, or proportional to another one. Thus, it is natural to expect bound functions to be reported in **asymptotic form**, in addition to or instead of non-asymptotic ones.

[1] Subject to the limits of computability theory.

[2] We use **bound function** or simply **bound** to refer to either a UBF or an LBF in closed form.

Due to the inherent incompleteness of automatic static analysis, sometimes the cost analyser is unable to compute a bound function, even if a non-trivial one exists. The most important properties of a cost analyser are its applicability, its precision, and its scalability. Concretely,

- The **applicability** refers to the class of programs for which the analyser can infer non-trivial bound functions.
- For those programs in the applicability range, the **precision** is the relation between the bound function computed by the analyser and the worst-case or best-case cost.
- The **scalability** refers to the ability of the analyser to retain performance levels, with the growth in the size of programs.

The applicability may be restricted by language features, like fields [6], or by semantic properties, like termination [5, 79, 140], or by the complexity class of the bound function, for instance, if it can only infer a bound function that is a linear [79, 35] or polynomial [65, 84, 73] function. Regarding the precision of the bound functions, some works focus on improving the functional precision [73]. However, the most important goal is to improve the **asymptotic** precision, that is to infer bounds that are in the exact asymptotic order of the program's cost.

There are several approaches to automatic static cost analysis. However, the most relevant ones for this thesis are the classical approach, and the automated amortised approach. In what follows we explain each of these approaches.

Classical Approach.

The classical approach to manually analyse the cost of a program [45, §2.2] is to first derive from the program a system of equations that defines its cost, and then solve those equations into a bound function. For instance, using the cost measure of the number of steps, the instructions, sequences and branches can be naturally represented as constants, sums and maxima, but each loop is usually written as a recurrence relation[3] (RR for short).

> "Recurrence relations arise frequently [...] through a direct mapping from a recursive representation of a program to a recursive representation of a function describing its properties." [118]

Interestingly, RRs only capture the essential properties of an algorithm, and these are the same for all of its implementations. After we have written the cost of a program as a system of RRs, in the manual approach one only needs solve them into a closed-form solution [61, 111, 58].

[3] Also called recurrence equations or difference equations.

Based on this approach, Cohen and Zuckerman [44] present an analyser for a simple imperative language, which is automatic only on the phase that generates RRs. The first fully *mechanical analysis of programs* was presented by Wegbreit [133]. He describes METRIC, a cost analyser for LISP programs that works in two phases: the first phase automatically transforms a LISP program into a system of RRs that describes its cost; and the second phase uses an automatic solver of RRs to derive a closed-form solution from the system.

Following Wegbreit, the classical approach to automatic static cost analysis splits the analysis in two independent phases: analysis and resolution.

1. The **analysis** transforms the program into a system of RRs. The construction of this phase may be different for different programming languages, for different kinds of cost model analysis, or different size measures used.
2. The **resolution** of a system of RRs computes a closed-form solution. Since RRs are a purely numerical representation, this phase is independent of the programming language or the cost model.

The benefit of the classical approach is that, by putting all the special features of the language, cost model, or size measure, into the first phase, the methods of the second phase are equally applicable to all languages or cost models. So, we can see a system of RRs as an *abstract program*, that serves as a common target for cost analysers of any language. Another benefit is that, if we have several programs implementing the same algorithm, and on each program we run an analyser for its respective language, then the systems of RRs tend to be (almost) the same. So, the first phase serves to clear the implementation details and to leave only the essence of the algorithm.

After Wegbreit's seminal article, many works have extended or adapted both phases of the classical approach. As these phases are independent, in what follows we briefly comment on the existing work on each one, by separate.

Generation of Recurrence Relations. Research on the first phase, on generating RRs, is concerned with dealing with the specific semantic features of each language, cost model, size measure, or the kind of cost function (worst, best or average case). Regarding programming languages, a lot of work exists on analysing declarative languages, because in these languages a program already is in a recursive representation. Analysers thus exist for functional [71, 132, 112, 62, 31, 131, 32, 51, 55], and logic programming languages [54, 99, 59]. In contrast, analysers for imperative or object-oriented languages are more scarce [57, 98, 86]. COSTA [10] is the first approach to analysing programs written in an object-oriented subset of JAVA Bytecode. The COSTABS [3] system is an analyser for the concurrent object-oriented language ABS. Regarding the cost model used, although most work has focused on models associated with time or time

complexity, there are also analysers for other models like heap memory [13], concurrent tasks [11] or energy [86]. Regarding the size abstraction, some works on functional languages use advanced type systems like dependent types [62] or sized types [119, 131]. Regarding the kind of cost function looked for, most works focus on inferring a UBF on the worst-case cost, but there are some that are able to infer an LBF on the best-case cost [95], or both UBF and LBF [112]. There are also some works that apply the classical approach to analysing average case analysis [71, 57]. These works have to deal with two difficult problems: the first one is to manage probability distributions on the data, which requires a special semantics [106]; the second is to consider the combinatorial properties of data structures [118, 58].

Recurrence Relations vs. Cost Relations. One major problem faced by analysers for this phase, starting from Wegbreit's [133], is dealing with the semantic gap between deterministic RRs and program semantics. In a program there usually are many execution paths, and these paths may be controlled by such guards or conditions that cannot be represented in the corresponding RR. As a result, for some values of the size of the input, the equations that correspond to different paths may overlap (i.e., are applicable on the same input). Several solutions exist for this problem. Some works [62, 51] directly use maxima operators to keep determinism. Karp proposes using Probabilistic RRs [82]. Rosendahl [112] proposes a more elegant solution, based on the theory of Abstract Interpretation [46], which is to consider a nondeterministic system of RRs as an *abstract program*, whose abstract semantics overapproximates the set of costs of the executions of the program. This solution is also used in COSTA [10], in which the system of nondeterministic RRs is called a **cost relation system** (CRS for short). The benefit of using a nondeterministic system of RRs or CRS as an abstract program, is that we can infer both UBFs and LBFs from the same abstract program.

Solving Recurrence Relations. Research on this phase deals with developing automatic methods to solve a system of RRs, or a nondeterministic CRS, into a bound function. Unlike the intense activity in the phase of generating RRs, research on this second phase has been slower and less extensive, because even *manually* solving a system of RRs into a closed-form solution is already too difficult. At the time of Wegbreit's article, there were no such automatic methods. The first methods were developed for computer algebra systems, like MACSYMA [78], MATHLAB [42], or MAPLE [57, 113]. Other works on methods to solve a system of RRs into closed-form solutions focus on handling special functions, like polynomials, or binomials [104, 105, 120]. Other recent works include [36, 39]. The PURRS system [27] implements some methods to solve or

approximate a given system of RRs.

However, all these works handle *deterministic* RRs, not the CRSs used, for instance, in COSTA [10]. There is too little work on solving CRSs or similar abstract programs. Rosendahl [112], who uses abstract programs to capture the cost, used program transformations inspired from ACE [96]. The PUBS system [5] is the first system that can handle a wide class of CRSs. It is based on viewing CRSs as an abstract programming language with an operational semantics, and using program analysis techniques to bound its worst case execution. In particular, it uses the connection between RR computation and termination analysis first described in [83]. The extension of PUBS described in [14] allows for inferring more precise UBFs or even LBFs.

Amortised Approach.

In the 60's and the 70's, the classical approach to cost analysis was (manually) applied to obtain accurate bound functions for many programs. However, in the early 80's some computer scientists noticed that when they applied the classical approach to analyse the performance of some programs that repeatedly operate on some data structures, like self-balancing binary search trees, they usually got an asymptotically imprecise UBF. The special feature of these programs is that, depending on the state of the data structure, each single operation could be expensive (have a high cost) or cheap (have a low cost). An analysis following the classical approach would consider all of them expensive, and thus would lead to asymptotically imprecise UBF if such scenario is not possible. This lead Robert Tarjan to develop an alternative approach to cost analysis.

Amortised cost analysis [129] is an approach to cost analysis in which the focus is not on the cost of one operation, but on the cost of a sequence of operations. This approach is based on the observation that, although one single operation can be expensive or cheap, in every sequence of operations there are enough cheap operations so as to *amortise* the cost of the expensive operations among the cheap ones. To prove that this *amortisation* happens, the methods of this approach use the following metaphor: they assume that the data structure stores some kind of savings, and they study the relation between the cost of an operation and its effect on the savings of the data structure. There are two main methods for amortised cost analysis that differ in how to represent those savings: in the banker's method it is represented as credit associated to each element in the data structure; whereas in the physicist's method, the savings are represented as a potential of the whole data structure. These methods are equivalent, so choosing one method or another depends on the program being analysed. The amortised approach is more precise than the classical approach,

in that it obtains an asymptotically precise UBF for the kind of programs mentioned above.

After Tarjan's seminal paper, amortised analysis has been applied to analyse the cost of operations on data structures, both in functional [103] and imperative [45, §17] languages. It application to an automatic cost analysis, called an *automated amortised analysis*, was introduced by Jost [75, 79]. His work considers a strict first-order functional language, and he presents an analysis, based on a type system for that language, that infers for each procedure in the program a couple of potential functions: the first one is a potential function over the input of the procedure, which gives a UBF on the cost of the procedure, and the second one is a potential function over the output of the procedure, which is used to pay for the cost of the subsequent operations on the output. Being based on types, this approach is very modular, since each function can be analysed separately.

The approach of Jost was later extended in several directions. Campbell [35] considers inferring, for a similar language, a bound on the stack space, for which he had to modify the type system. Rodriguez [109] applies the automated amortised analysis to an object-oriented language, for which it is necessary to handle inheritance. The system RAML [73] is able to infer multivariate polynomial UBFs, by considering a wide set of type-directed norms, which includes products of polynomials on the lengths of two lists, or introspective measures like the sum of the lengths of the lists in a matrix. Jost et al. [80] consider a strict higher-order functional language, whose analysis requires taking into account the cost of evaluating each partial function application. Scherer and Hoffmann [117] consider the case in which the cost depends on those arguments already stored in the closure of a partial function application. Simões et al. [122] consider a language with lazy evaluation, for which they need to handle *thunks*, that is, function calls whose evaluation is suspended. Hoffmann and Shao [74] extend the analysis to an imperative language with arrays and integer arithmetic operations. Atkey [23] presents a method to analyse the cost of linked data structures, like lists, inspired on the banker's method and based on separation logic [107].

2 Applications of Cost Analysis

Cost analysis has many potential applications. In hardware development, cost analysis is used to analyse and verify the worst-case execution time of processors [137], especially relevant for real-time systems. Nowadays, there is also an interest in energy consumption [86] for embedded systems.

There are also different applications for cost analysis in parallel comput-

ing (multiprocessor or distributed). One application is that of load balancing and granularity control of programs, that is, to estimate the optimal way to divide a computation into parallel tasks and how to split those tasks among the processors. One way to do this is to use an analysis to estimate a good work load for each processor [95]. A more flexible technique is to use *autonomous mobile programs (AMP)* [55], where each AMP analyses the cost of its pending computations, and decides by itself to migrate to another processor with smaller workload. In a distributed computation, as the communication network between nodes can become a performance bottleneck, it is also important to estimate how many messages are sent across the network [12]. Also, if we write our programs in a concurrent language, we may want to know the peak number of active concurrent tasks [11], as this is the maximum parallelism that can be exploited for this program. Cost analysis can also be applied in the context of cloud computing [22], in which a company provides virtual computing capabilities by hour. In this context, cost analysis serves as a tool to estimate, before the transaction, the price of the computation to purchase.

Programmers can also use cost analysis when developing programs, by writing assertions on the expected cost, and letting the cost analyser verify these assertions: such an approach to resource usage verification [94] is integrated in the CIAO compiler [69]. Also cost analysis can be applied when deploying and running a program, in the context of resource bound certification [50, 9]. This is based on the idea of attaching to the program a proof that witnesses that it meets a specification, a general technique known as constructive program verification [135].

Liang [92] applies cost analysis to query optimisation of logic programs. Cost analysis also serves to automatically adapt a program to the hardware platform in which it runs: in the PETABRICKS [20] language, a programmer can write several procedures to perform a given task, each one implementing a different algorithm. The compiler for this language and the *autotuner* automatically choose the most efficient procedure for a given architecture and size of data. Then a cost analyser can be used to guide the *autotuning*.

3 Thesis Objectives

The field of this dissertation is automatic static cost analysis using the classical approach, which is based on the two phases, one for generating the abstract programs, and another for solving these abstract programs into bound functions. The main issues in this field are the applicability and the precision, especially the asymptotic one, of these analysers.

The main goal of this dissertation is to improve the precision and applica-

bility of cost analysers based on this approach, to make their precision similar to those based on amortised analysis. In particular, we are interested in improving the techniques on the second phase, to go ***from abstract programs to precise asymptotic closed-form bounds***. In short, we seek to build[4] an analyser that infers more asymptotically precise UBFs for more programs.

The main challenge is the precision gap between the classical and amortised approaches. Although it is known that there are programs for which analysers based on the latter infer more precise UBFs than those based on the former, it is not clearly explained why that happens, nor a criterion to know for which programs. This lead to some misconceptions about the amortised approach, so that some texts [103] use the notion of *amortised cost* in contraposition to *worst-case* cost, as if they were different properties of programs instead of different methods of analysis. One of this misconceptions is that amortised cost analysis is only applicable to data structures, not to simple algorithms or loops. In fact, the methods of automated amortised analysis [79] define the potential as a measure related to data structures or constructors.

As a result of these misconceptions, for a long time, it was assumed that in order to infer a precise bound function it was necessary to use the techniques of the amortised approach. The idea that it was possible to infer precise bounds *without* using the techniques of amortised analysis, was first observed in the context of the SPEED project [65]. Research on this project was focused on cost analysis[5] of imperative programs, consisting in nested loops in which inner and outer loops share some counter variables. For these programs, analysers based on the classical approach, like COSTA [7, 1], as well as other analysers that do not follow the classical approach, like [84], infer an asymptotically imprecise UBF, or even fail to infer any UBF at all. Instead, the members of the SPEED project achieve the automatic inference of a precise UBF using techniques related to termination analysis of loops [65, 64, 66, 140]. This suggests that it is possible to achieve what an amortised analysis does *without* using the techniques of amortised analysis. The starting challenge for this thesis was to develop cost analyses techniques so as to infer UBFs as precise as those of the amortised approach, while keeping the separation of phases as in the classical approach. This separation is essential to obtain a cost analyser that is generic on the programming language, parametric on the cost models, and adaptable to compute UBFs or LBFs either in asymptotic or non-asymptotic form.

[4] That is, learn what problems need to be considered and how to solve them in order to build.

[5] The SPEED project only considered one cost model, the number of iterations of loops, so their analyses were called *bound* or *reachability bound analyses* [140].

4 Contributions and Outline

As a background, in Chapter 2 we present COSTA and PUBS. COSTA is a cost and termination analyser for JAVA Bytecode programs, and PUBS is a CRS solver, which follow the division of work in the classical approach. The contributions of this thesis consist in improvements and extensions to the different parts of COSTA and PUBS, but they can be applied to any other approach with similar architecture. In Chapters 4-6 we describe the contributions of this thesis, and relate them to the supporting articles. Each of these chapters also includes a discussion of related works and a discussion on further points that can be found in the article. Finally, in Chapter 7 we draw our conclusions and discuss possible future work. Next we briefly describe the main contributions.

An Asymptotic Transformation. Bound functions can be big and intricate expressions, difficult to read or process. Asymptotic bounds are smaller and simpler, and they succinctly describe how the cost scales on the size of the input of the program. Unfortunately, many cost analysers do not infer asymptotic bounds. Our first contribution is an algorithm to transform bound functions to asymptotic form, and a scalable asymptotic CRSs solver that is based on this transformation. This contribution was presented in the following article:

> ELVIRA ALBERT, DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, SAMIR GENAIM, AND GERMAN PUEBLA. Asymptotic Resource Usage Bounds. In Zhenjiang Hu, editor of the Proceedings of *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 5904 of *Lecture Notes in Computer Science*, pages 294–310. Springer, December 2009

In this dissertation, the contribution is described in Chapter 3, and the article is attached at Page 195.

A Value Analysis for Nonlinear Operations. In order to infer precise bounds, COSTA uses value relations, as conjunctions of linear constraints, that approximate the values that program variables can take at runtime. Our second contribution studies the limitations of this approximation step, in particular its inability to model nonlinear arithmetic operations, and develop a technique to overcome these limitations, in a scalable way. This contribution was presented in the following article:

> DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Handling Non-linear Operations in the Value Analysis of COSTA. *Electronic Notes in Theoretical Computer Science*, 279(1):3–17, 2011. Proceedings of Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)

In this dissertation, the contribution is described in Chapter 4, and the article is attached at Page 217.

An Approach to handle the Output-Cost Codependency in Cost Analysis. In most cost analysers, the cost of a program is captured only in terms of the input of the program. However, in some programs another codependency appears between the cost and the output of a procedure, and any cost analysis that ignores this codependency necessarily obtains a UBF that is asymptotically imprecise. Our third contribution gives a detailed example of this codependency, and explains how it affects the precision of the analysis of COSTA. We then present an approach to cost analysis that keeps this codependency in order to obtain a precise UBF. This approach is based in the automatic techniques of logical analysis of programs [34], namely satisfiability modulo theory (SMT) and quantifier elimination (QE) [89, §9]. Importantly, we also expose a strong relation between the output-cost codependency and amortised analysis. This contribution was presented in the following article:

> DIEGO ESTEBAN ALONSO-BLAS AND SAMIR GENAIM. On the Limits of the Classical Approach to Cost Analysis. In Antoine Miné and David Schmidt, editors of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 7460 of *Lecture Notes in Computer Science*, pages 405–421. Springer, September 2012

In this dissertation, the contribution is described in Chapter 5, and the article is attached at Page 233.

A Precise and Practical Method to Solve CRSs. Our last contribution is a new method to solve CRSs (abstract programs) into UBFs, that obtains, for many abstract programs, UBFs that are asymptotically accurate. This method is based on complete and scalable techniques for quantifier elimination and satisfiability in the theory of linear real arithmetic, which makes the key for scalability. This contribution is presented in the following article:

> DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Precise Cost Analysis via Local Reasoning. In Dang Van Hung and Mizuhito Ogawa, editors of the Proceedings of *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 8172 of *Lecture Notes in Computer Science*, pages 319–333. Springer, October 2013

In this dissertation, the contribution is described in Chapter 6, and the article is attached at Page 253.

2 | The COSTA System

This thesis presents techniques to compute precise asymptotic bounds from nondeterministic abstract programs. We developed these techniques in the context of COSTA. This chapter describes the architecture and the inner workings of COSTA that are necessary to explain our contributions.

1 The Architecture of COSTA

COSTA [7] is an automatic static analyser for JAVA Byte-Code (JBC for short), a low-level object-oriented language [93]. It takes as input a program written in JBC, and can prove its termination or compute a bound on its cost, with respect to the given cost model. This bound can be either a UBF on its worst-case cost, or a LBF on its best-case cost. These bounds are numeric functions on the size of the input, and they are in closed-form, i.e., deterministic and nonrecursive.

The architecture of COSTA is illustrated in Figure 2.1. The rectangular boxes with rounded corners represent its main components (transformations, analyses, etc.), and the rectangular boxes represent intermediate information that is passed between these components. The ellipses are annotations that highlight the contributions of this thesis. The rest of this chapter describes the inner workings of COSTA, by applying each of the following components on a simple program that we introduce in Example 2.1:

- The **Rule-Based Representation** (RBR for short) is obtained by a declarative compilation of the Control Flow Graph (CFG for short) of the input JBC program. An RBR program is a set of rules that define several procedures, with one rule per basic block in the program, and one procedure per method, loop or branching statement. The RBR provides a uniform representation of the control-flow as guarded rules and interprocedural calls, in which all iterations are performed via recursive calls. Section 2.2 discusses the RBR and the result of the decompilation for our example.
- The **Abstract Cost Rules** (ACR for short) program is obtained from the RBR by abstract compilation: a **size abstraction** maps each data structure to a size variable and each operation to a constraint between size variables; and a **cost model** maps each RBR instruction to a **cost annotation** that models its cost. Section 2.3 describes the syntax and semantics of ACR programs, and the abstract compilation for our example.
- The **Cost Relation System** (CRS for short) is obtained from the ACR by removing the output variables from each inter-procedural call, and replacing

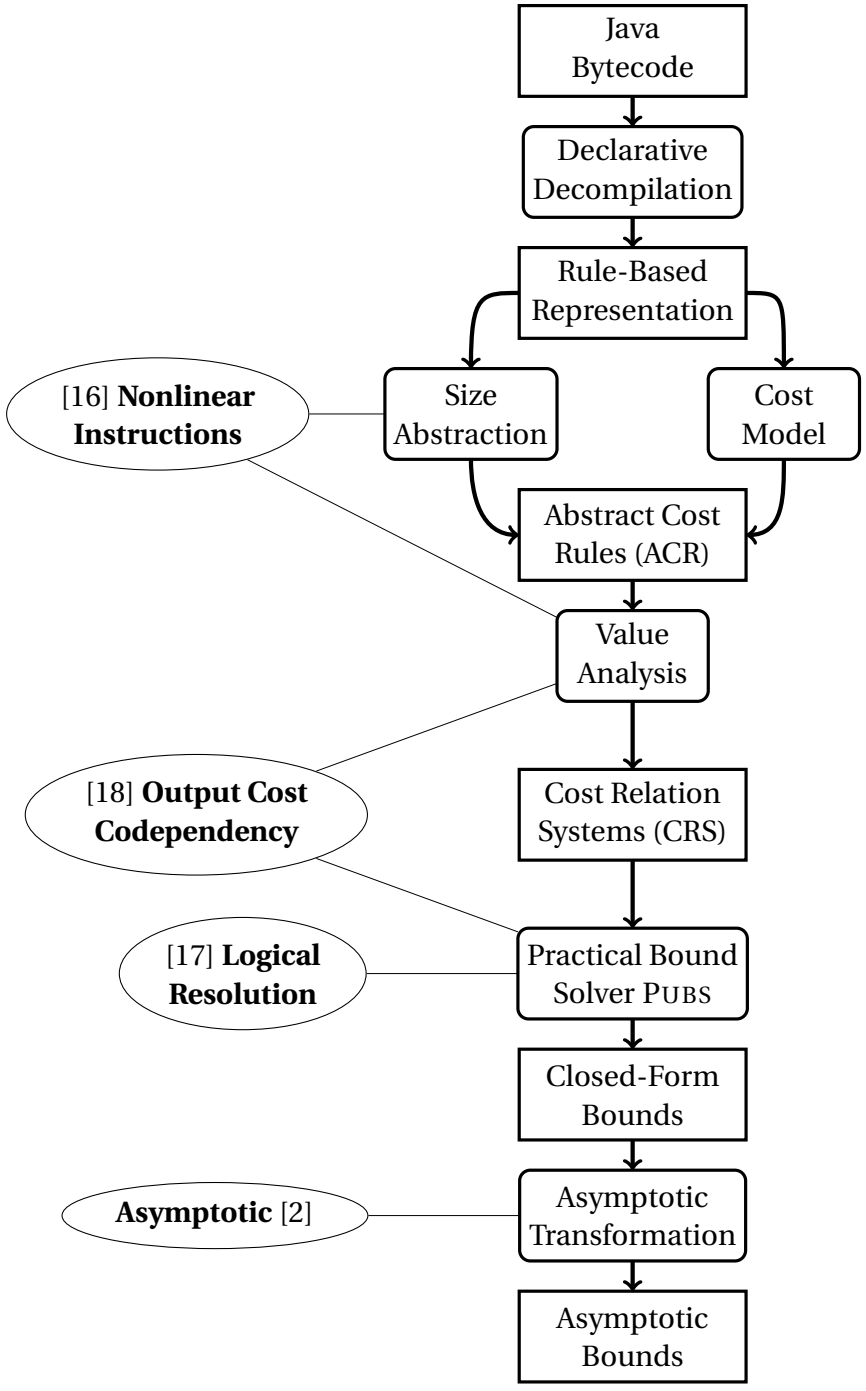


Figure 2.1: The architecture of COSTA: analyses, transformations, intermediate information formats, and the contributions of this thesis.

those output variables with a postcondition formula computed by a **value analysis**. The CRS thus represents the relation between the input and the cost of each procedure. Section 2.4 discusses the syntax and semantics of the CRS, the ACR to CRS transformation, and the CRS generated for our example.

- The final step is to solve the CRS into UBFs or LBFs. For this, COSTA relies on its subsystem PUBS (Practical Upper Bound Solver) [5], which takes as input a CRS and computes its corresponding UBFs or LBFs. Section 2.5 describes the solving procedure of PUBS, and applies it to our example.

Note that although COSTA was originally designed to analyse JBC programs, it is easy to adapt this architecture to other languages.

Example 2.1. The JAVA code of our running example for this chapter is depicted in Figure 2.2. It is an implementation of the insertion sort algorithm. The method `inSort` sorts the array elements from position 0 to position `last` -1, using the auxiliary method `insert`. We have compiled this method to JBC and analysed it with COSTA to obtain bounds on the number of read and write accesses to the array `data`. We assume that the two read accesses to `data[j` -1`]` at Lines 10 and 11 are optimised by the compiler to one access using an auxiliary local variable (this will be reflected in the RBR). For `insert(data,i)`, COSTA reports the UBF $2 * \mathtt{nat}(i-1)+3$, which can be explained as follows: the array accesses at Lines 8 and 14 contribute 2; the `while` loop contributes $2 * \mathtt{nat}(i-1)$, which is the product of the number of array accesses 2 by the number of iterations $\mathtt{nat}(i-1)$; and, finally, the access in the loop guard when it evaluates to *false* contributes 1. Here, $\mathtt{nat}(x)$ means $\max(x,0)$. For `inSort(data,last)`, COSTA reports the UBF $\mathtt{nat}(last-1) * (2 * \mathtt{nat}(last-1)+3)$. The factor $\mathtt{nat}(last-1)$ is the number of iterations of the `for` loop. The second factor is the worst-case cost of all iterations, which is as the UBF of `insert` except that i is replaced by $last-1$, i.e., by the maximum value that the loop counter `i` can take. ■

Next we describe the steps that COSTA follows to infer the UBF of the above example. In order to abridge the presentation, we have simplified the intermediate results of the analysis, without affecting the correctness of these bounds. In addition, we skip many details that are not necessary to present our contributions in the subsequent chapters. For a comprehensive description of COSTA the reader may refer to [7, 1].

2 Rule-Based Representation

COSTA starts by constructing a CFG for the JBC program. The CFG for the (JBC of the) methods `inSort` and `insert` is depicted in Figure 2.3. Note that,

```
1   // Precondition: 0 <= last <= data.length
2   void inSort(double[] data, int last) {
3       for (int i=1; i<last; i++)
4           insert(data,i);
5   }
6
7   void insert( double[] data, int i){
8       double x = data[i];
9       int j=i;
10      while (j > 0 && data[j - 1] < x) {
11          data[j] = data[j - 1];
12          j--;
13      }
14      data[j] = x;
15  }
```

Figure 2.2: JAVA code of the `inSort` method.

for simplicity, the exceptional behaviour is excluded. A square box represents a basic block, which is a sequence of JBC instructions without branching. A diamond represents a branching point. A circled point marks a method end or a loop exit. For the sake of simplicity, we omit instructions in the basic blocks and show only the names of each procedure and the guards of each branch.

The CFG is split into five subgraphs or **procedures**: one for the method `inSort`, one for the `for` loop, one for the method `insert`, and two for the `while` loop (Lines 10 to 13). The `while` loop has two procedures because the condition $\texttt{data}[\texttt{j}-1] < \texttt{x}$ (Line 10) is evaluated only if $\texttt{j} > 0$ is *true*. Each subgraph (or procedure) has one or more control path. A control path represents a jump to a basic block, with a **guard** to specify in what cases this jump can be taken. The procedures for the methods `inSort` and `insert` have one control path with the *true* guard. Procedures that correspond to loops have two control paths: one to exit the loop and one to enter its body. For instance, in the procedure of the `for` loop the exit path is labelled with $\neg(\texttt{i} < \texttt{last})$, and the other one is labelled with $\texttt{i} < \texttt{last}$.

COSTA relies on points-to analysis [125] to resolve virtual method invocations when generating the CFG, since the method to be executed is known only at runtime. In addition, COSTA extracts and isolates each loop as if it were a method, when possible, in which case the exit path is empty instead of including the instructions executed after the loop. Loop extraction [136] is a technique that COSTA uses to allow compositional analyses [7].

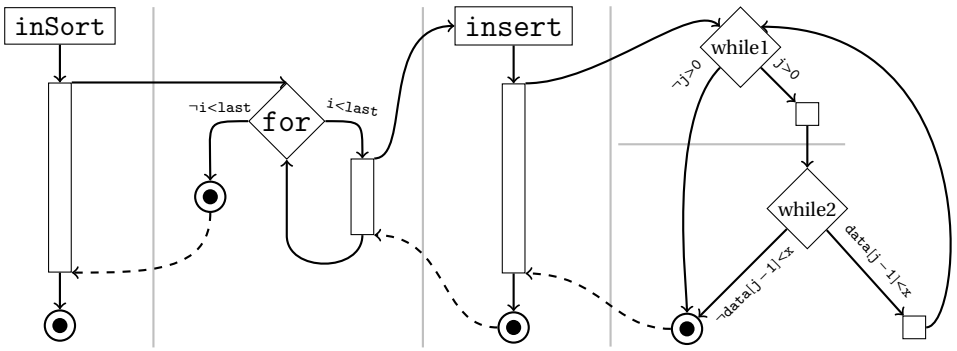


Figure 2.3: Control Flow Graph of the `inSort` method.

Translation to the RBR.

The next step of COSTA is to transform the CFG into an RBR program, that consists of rules of the form

$$p(\bar{x},\bar{y}) \leftarrow \{g\}, b_1, \ldots, b_n.$$

where p is a procedure name, $\bar{x}$ is a list of its input variables, $\bar{y}$ is a list of its output variables, g is a guard, and each b_i is an RBR instruction.

The guard is a logical formula defined over the input variables, and will be omitted if it is equal to *true*. An RBR instruction is either a call $q(\bar{w},\bar{z})$ to a procedure q; or a non-branching instruction that typically corresponds to a JBC instruction. We restrict ourselves to assignments of the form $\tau := e$, where e is an arithmetic expression over variables and array accesses, and τ is either a variable x or an array position $x[i]$. Note that the RBR of COSTA includes many other instructions, e.g., field accesses, object and array creation, etc. However, they are not necessary for explaining our contributions, so we omit the corresponding details.

To translate a CFG to an RBR, each procedure of the CFG is mapped to a procedure in the RBR. Each control path in the CFG is transformed into a rule in the RBR, with a guard that is the same as that of the path. Each interprocedural jump between blocks in the CFG is transformed to a corresponding procedure call. Each instruction in the JBC program is transformed to a corresponding RBR instruction. Variables in the RBR are local to the rule in which they appear, and they typically correspond to either variables in the JAVA program, or positions in the operands stack of the JVM.

Example 2.2. The RBR of our running example is depicted in Figure 2.4. It contains eight rules that define five procedures, which correspond to the procedures and control paths of the CFG of Figure 2.3.

$inSort(\langle data, last\rangle, \langle\rangle) \leftarrow$
  $i := 1,$
  $for(\langle data, last, i\rangle, \langle\rangle).$

$for(\langle data, last, i\rangle, \langle\rangle) \leftarrow$
  $\{\neg i < last\}.$
$for(\langle data, last, i\rangle, \langle\rangle) \leftarrow$
  $\{i < last\},$
  $insert(\langle data, i\rangle, \langle\rangle),$
  $i := i + 1,$
  $for(\langle data, last, i\rangle, \langle\rangle).$

$insert(\langle data, i\rangle, \langle\rangle) \leftarrow$
  $x := data[i],$
  $j := i,$
  $while_a(\langle data, x, j\rangle, \langle j\rangle),$
  $data[j] := x.$

$while_a(\langle data, x, j\rangle, \langle j\rangle) \leftarrow$
  $\{\neg j > 0\}.$
$while_a(\langle data, x, j\rangle, \langle j\rangle) \leftarrow$
  $\{j > 0\},$
  $jj := j - 1,$
  $y := data[jj],$
  $while_b(\langle data, x, j, y\rangle, \langle j\rangle).$

$while_b(\langle data, x, j, y\rangle, \langle j\rangle) \leftarrow$
  $\{\neg y < x\}.$
$while_b(\langle data, x, j, y\rangle, \langle j\rangle) \leftarrow$
  $\{y < x\},$
  $data[j] := y,$
  $j := j - 1,$
  $while_a(\langle data, x, j\rangle, \langle j\rangle).$

Figure 2.4: Rule-Based Representation program for method `inSort`.

Procedures *inSort* and *insert* correspond to the JAVA methods of Figure 2.2, note that their input and output parameters coincide. The rule of *inSort* initialises variable i of the `for` loop and then calls procedure *for*. The rule of *insert* contains one instruction that corresponds to the array access at Line 8 of Figure 2.2, another one to initialise the variable j of the `while` loop, a call to the $while_a$ procedure, and one that corresponds to the array update at Line 14.

The RBR procedures *for*, $while_a$ and $while_b$ correspond to the `for` and `while` loops of the JAVA program. Each procedure has two rules: one to exit the loop, and one to enter its body. It can be seen that looping behaviour is done using tail-recursion in the RBR. Note that the two accesses to `data[j-1]` in the `while` loop of method `insert` are optimised to one in the RBR, by storing the corresponding value in the auxiliary variable y (in the second rule of $while_a$). ■

RBR Semantics.

A trace semantics for RBR programs is presented in [10]; we skip the details since they are not important to explain our contributions. Informally, assuming that each RBR instruction is assigned some cost (using some cost model), the cost of an RBR trace t, denoted $rbrcost(t)$, is defined as the sum of all such costs contributed by the instructions executed within t. A function p^+ (resp. p^-) is a UBF (resp. LBF) for an RBR procedure p, if and only if for any (possibly partial) trace t that corresponds to executing p starting from an initial input of size $\bar{v}$,

we have $p^+(\bar{v}) \geq rbrcost(t)$ (resp. $p^-(\bar{v}) \leq rbrcost(t)$).

The correctness of the JBC to RBR transformation relies on that there is a one-to-one correspondence between JBC and RBR traces. This means that the cost described above can be seen as the cost of the original JBC program as well.

3 Abstract Cost Rules

The next step of COSTA is the abstract compilation of the RBR program into an ACR program, that consists of rules of the form:

$$p(\bar{x}^\alpha, \bar{y}^\alpha) \leftarrow a_0, \ldots, a_n.$$

where p is a procedure name; $\bar{x}^\alpha$ and $\bar{y}^\alpha$ are respectively lists of its input and output (abstract) variables; and each a_i is an ACR instruction that can be either a procedure call $q(\bar{w}^\alpha, \bar{z}^\alpha)$, a linear constraint over the rule's variables (equality or non-strict inequality), or a cost annotation of the form acquire(e), which indicates that, at the corresponding program point, we accumulate e to the cost.

The abstract variables in an ACR rule are integer variables, and they represent the size of corresponding RBR variables. COSTA uses different size measures depending on the JBC type of the corresponding variable: the size of an `int` variable is its value; the size of an array is its length, the size of an object reference is its path-length [123], and variables of non-integer basic types are abstracted to "free" variables that represent any value, and thus not tracked at all. If the cost depends on these "free" variables, COSTA will not be able to infer a UBF for the ACR program.

The abstract compilation compiles each RBR rule "$p(\bar{x}, \bar{y}) \leftarrow \{g\}, b_1, \ldots, b_m$" into a corresponding ACR rule "$p(\bar{x}^\alpha, \bar{y}^\alpha) \leftarrow a_0, \ldots, a_n$" where: the abstract variables $\bar{x}^\alpha$ and $\bar{y}^\alpha$ correspond to the sizes of $\bar{x}$ and $\bar{y}$ respectively; the guard g is compiled to a corresponding linear constraint a_0, over the abstract input variables $\bar{x}^\alpha$, that overapproximates its behaviour; and each b_i is compiled to one or more consecutive a_j as follows:

- **Size abstraction.** If b_i is a call $q(\bar{w}, \bar{z})$ then it is compiled to $q(\bar{w}^\alpha, \bar{z}^\alpha)$; if it is an assignment of the form $x := e$ where e is a linear expression; then it is compiled to $x^\alpha = e^\alpha$ where e^α is obtained from e by replacing its variables by their corresponding abstract versions; otherwise it is compiled to the constraint *true*, which has no effect on the execution. The size abstraction uses a Static Single Assignment (SSA) transformation [21, §19] to simulate the effect of updates with linear constraints. Note that since COSTA supports several other RBR instructions, e.g., field access, the actual size abstraction

$inSort(\langle l_0\rangle,\langle\rangle) \leftarrow$
  $i_2 = 1$,
  $for(\langle l_0, i_2\rangle,\langle\rangle)$.

$for(\langle l_0, i_0\rangle,\langle\rangle) \leftarrow$
  $i_0 \geq l_0$.
$for(\langle l_0, i_0\rangle,\langle\rangle) \leftarrow$
  $i_0 + 1 \leq l_0$,
  $insert(\langle i_0\rangle,\langle\rangle)$,
  $i_2 = i_0 + 1$,
  $for(\langle l_0, i_2\rangle,\langle\rangle)$.

$insert(\langle i_0\rangle,\langle\rangle) \leftarrow$
  acquire(1),
  $while_a(\langle i_0\rangle,\langle\rangle)$,
  acquire(1).

$while_a(\langle j_0\rangle,\langle\rangle) \leftarrow$
  $j_0 \leq 0$.
$while_a(\langle j_0\rangle,\langle\rangle) \leftarrow$
  $j_0 \geq 1$,
  acquire(1),
  $while_b(\langle j_0\rangle,\langle\rangle)$.
$while_b(\langle j_0\rangle,\langle\rangle) \leftarrow true$.
$while_b(\langle j_0\rangle,\langle\rangle) \leftarrow$
  acquire(1),
  $j_2 = j_0 - 1$,
  $while_a(\langle j_2\rangle,\langle\rangle)$.

Figure 2.5: Abstract Cost Rules (ACR) program for method `inSort`.

is more elaborated. However, these details are not important for our contributions since we will assume a given ACR program independently from where it comes.

- **Cost annotation.** The selected **cost model** is applied to b_i to generate a cost annotation of the form acquire(e) that describes its cost. We omit these annotations from the ACR when $e = 0$. COSTA has several cost models, e.g., for counting the number of executed instructions, calls to a specific method, or occupied memory.

The abstract compilation ends by removing from the ACR program those variables that do not affect the cost [8]. This results in concise ACR programs that are more efficient to analyse.

Example 2.3. The ACR program obtained by applying the abstract compilation to the RBR program of Figure 2.4 is depicted in Figure 2.5. The size abstraction maps the RBR variables *last*, i, and j, to the ACR variables l, i and j that appear with subscripts introduced by SSA (see explanation in the next paragraph). The RBR variables x, y, *data*, and the output variables of procedures $while_a$ and $while_b$ are omitted because they do not affect the cost. Following Example 2.1, we use a cost model that counts the number of array accesses.

The relation between each RBR rule and its corresponding ACR rule is clear. Note that the assignment $i := i + 1$ in the second RBR rule of procedure *for* was compiled to the constraint $i_2 = i_0 + 1$. Here i_0 and i_2 represent those values of variable i before and after executing the instruction. Note also the use of the cost annotations acquire(1) to account 1 as the cost of each array access. ■

ACR Semantics.

Let us briefly summarise the important details of the semantics of ACR programs [18, 10]. A state s takes the form $\langle \psi, \bar{a} \rangle$, where $\bar{a}$ is a sequence of ACR instructions pending for execution, and ψ is a constraint over the variables in $\bar{a}$ and possibly other existentially quantified variables. The store ψ collects all constraints encountered during an execution. An execution starts from an initial state $\langle \bar{x} = \bar{v}, p(\bar{x}, \bar{y}) \rangle$, where $\bar{v}$ is a sequence of integers, and proceeds according to the following rules:

$$\frac{q(\bar{x},\bar{y}) \leftarrow \bar{a}' \in P}{\langle \psi, q(\bar{x},\bar{y}) \cdot \bar{a} \rangle \overset{0}{\rightsquigarrow} \langle \psi, \bar{a}' \cdot \bar{a} \rangle} \qquad \frac{\psi \wedge \varphi \not\models \mathit{false}}{\langle \psi, \varphi \cdot \bar{a} \rangle \overset{0}{\rightsquigarrow} \langle \psi \wedge \varphi, \bar{a} \rangle} \qquad \frac{\mathit{eval}(e,\psi) = v \geq 0}{\langle \psi, \mathsf{acquire}(e) \cdot \bar{a} \rangle \overset{v}{\rightsquigarrow} \langle \psi, \bar{a} \rangle}$$

These rules define a transition relation $s_1 \overset{v}{\rightsquigarrow} s_2$, meaning that there is a transition from s_1 to s_2 that costs v units. The rule on the left handles procedure calls, it (nondeterministically) selects a rule from the program P that matches the call, and adds its instructions $\bar{a}'$ to the sequence of pending instructions. Variables in $\bar{a}'$ (except $\bar{x} \cup \bar{y}$) are renamed such that they are different from the variables in $\bar{a}$ and ψ. The rule in the middle handles constraints by adding them to the store, if the resulting state is satisfiable. The rule on the right handles cost annotations, it evaluates e to a non-negative value v and labels the corresponding transition with v.

The execution stops when no rule is applicable, which happens when the execution reaches a) a final state $\langle \psi', \epsilon \rangle$ where ϵ is the empty sequence; or b) a blocking state $\langle \psi', \varphi \cdot \bar{a} \rangle$ where $\varphi \wedge \psi' \models \mathit{false}$. A trace t is a finite or infinite sequence of states, in which there is a valid transition between each pair of consecutive states. Traces that end in a final state and infinite traces are called complete. Namely, we exclude traces that end in a blocking state because such traces correspond to no RBR trace. The cost of a trace t, denoted $\mathit{acrcost}(t)$, is defined as the sum of all cost labels in its transitions. A function p^+ (resp. p^-) is a UBF (resp. LBF) for procedure p, if for any input $\bar{v}$ and complete trace t that starts in $\langle \bar{x} = \bar{v}, p(\bar{x}, \bar{y}) \rangle$, we have $p^+(\bar{v}) \geq \mathit{acrcost}(t)$ (resp. $p^-(\bar{v}) \leq \mathit{acrcost}(t)$).

The correctness of the RBR to ACR transformation relies on the idea that any RBR trace has a corresponding ACR trace with the same cost. Thus, a valid UBF or LBF for the ACR program is also valid for the RBR program.

4 Cost Relation Systems

In the next phase, the ACR program is transformed into a CRS, which consists of equations of the form:

$$C(\bar{x}) = e + D_1(\bar{y}_1) + \cdots + D_r(\bar{y}_r) \;, \varphi$$

where $C, D_1, \ldots, D_r$ are relation symbols (like procedure names); $\bar{x}, \bar{y}_1, \ldots, \bar{y}_r$ are variables; φ is a conjunction (often written as a set) of linear constraints over these (and maybe other) variables; and e is a **cost expression** that adheres to the following grammar:

Cost Expression	$\mathsf{exp} ::= \mathsf{bexp} \mid \mathsf{oexp}$
Basic Expression	$\mathsf{bexp} ::= q \mid \mathsf{nat}(l) \mid \log_m(\mathsf{nat}(l)+1) \mid m^{\mathsf{nat}(l)}-1$
Compound Expression	$\mathsf{oexp} ::= \mathsf{exp}+\mathsf{exp} \mid \mathsf{exp}*\mathsf{exp} \mid \max(\mathsf{exp}_1, \ldots, \mathsf{exp}_n)$

in which $q \in \mathbb{Q}^+$, $m > 1 \in \mathbb{Z}^+$, l is a linear expression over the variables that appear in the equation, and $\mathsf{nat}(l) = \max(l, 0)$. Intuitively, the above equation states that the cost of C, with respect to the input $\bar{x}$, is e plus the sum of the costs of each D_i with respect to the input $\bar{y}_i$. The linear constraint φ specifies the values of $\bar{x}$ for which the equation is applicable, and defines relations among the different variables. Note the declarative nature of this intuition. Since CRSs originate from ACR programs, we may think of $C, D_1, \ldots, D_r$ as (nondeterministic) procedures, and say that C calls $D_1, \ldots, D_r$.

All equations that share the same left-hand side, e.g., $C(\bar{x})$, define the **cost relation** (CR for short) $C(\bar{x})$. Thus, a CRS consists of several CRs. When a CR uses only one relation symbol, formally $D_i = C$ for all $1 \leq i \leq r$, we call it **standalone** CR, that is, it depends on no other CR.

The ACR to CRS transformation maps each ACR procedure $p(\bar{x}, \bar{y})$ to a CR $p(\bar{x})$, defined over the same input $\bar{x}$, but without the output $\bar{y}$. This is done by transforming each ACR rule "$p(\bar{x}, \bar{y}) \leftarrow a_1, \ldots, a_n$" to a corresponding equation "$p(\bar{x}) = e + q_1(\bar{w}_1) + \cdots + q_n(\bar{w}_n), \varphi$" as follows:

1. the cost expression e is computed as the sum of all expressions e_i in instructions $\mathsf{acquire}(e_i)$ that appear in the ACR rule. Thus, each e_i must be a valid cost expression;
2. each call $q_j(\bar{w}_j, \bar{z}_j)$ in the ACR rule is transformed to a call $q_j(\bar{w}_j)$ in the equation, with the same input but without the output; and
3. the constraint φ is a conjunction of all linear constraints in the ACR rule.

In the second step above, when $q_j(\bar{w}_j, \bar{z}_j)$ has a non-empty list of output variables, removing these variables may result in a loss of information that is crucial for inferring a corresponding UBFs. To reduce the effect of this loss, COSTA uses an inter-procedural **value analysis**[1] to compute a postcondition for each ACR procedure, and then adds this postcondition to φ to compensate on the removal of the output variables. This postcondition is a conjunction of linear

[1]The notion of value analysis is related to that of size analysis [121]. A size analysis infers relations not only between the values of numeric variables, but also between the sizes of data structures.

$$
\begin{array}{rcll}
\mathit{inSort}(l_0) & = & \mathit{for}(l_0, i_2) & \{i_2 = 1\} \\
\mathit{for}(l_0, i_0) & = & 0 & \{i_0 \geq l_0\} \\
\mathit{for}(l_0, i_0) & = & \mathit{insert}(i_0) + \mathit{for}(l_0, i_2) & \{i_0 + 1 \leq l_0, i_2 = i_0 + 1\} \\
\mathit{insert}(i_0) & = & 2 + \mathit{while}_a(i_0) & \{\} \\
\mathit{while}_a(j_0) & = & 0 & \{j_0 \leq 0\} \\
\mathit{while}_a(j_0) & = & 1 + \mathit{while}_b(j_0) & \{j_0 \geq 1\} \\
\mathit{while}_b(j_0) & = & 0 & \{\} \\
\mathit{while}_b(j_0) & = & 1 + \mathit{while}_a(j_2) & \{j_2 = j_0 - 1\}
\end{array}
$$

Figure 2.6: Cost Relation System (CRS) for the `inSort` method.

constraints that describe the relation between input and output variables, in any execution of the corresponding procedure. It is worth emphasising that postconditions in COSTA are *linear*, and thus they cannot model nonlinear relations. Our contribution of Chapter 4 improves in this direction.

Example 2.4. The CRS generated from the ACR program of Figure 2.5 is depicted in Figure 2.6. It has eight equations that define five CRs. The cost expression in each equation coincides with the cost annotations of its corresponding ACR rule: 2 for *insert*, 1 in the recursive rules of while_a and while_b, and 0 elsewhere. The calls in each equation are as those in the ACR, and the constraints in each equation are those which appear in the corresponding ACR rule. Note that, in this example, there was no need to perform value analysis because the ACR procedures in Figure 2.5 have no output variables. ■

CRS Semantics.

The semantics of the CRS is based on the notion of evaluation trees [5]. Let us denote a (possibly infinite) tree by $\mathit{node}(q, \langle T_1, \ldots, T_k \rangle)$, where $q \in \mathbb{Q}^+$ is the value of the root and $T_1, \ldots, T_k$ are subtrees. Given a CR C and a concrete input $\bar{v}$, we say that $\mathit{node}(v_e, \langle T_1, \ldots, T_k \rangle)$ is an **evaluation tree** for $C(\bar{v})$ if and only if there exists an equation "$C(\bar{x}) = e + \sum_{j=1}^{k} D_j(\bar{y}_j), \varphi$" and an assignment σ, for the variables of that equation, such that:

1. $\sigma(\bar{x}) = \bar{v}$ and σ is a satisfying assignment for φ;
2. e is evaluated to v_e when considering the assignment σ; and
3. each T_i is an evaluation tree for $C(\sigma(\bar{y}_i))$.

Intuitively, when viewing C as a procedure, an evaluation tree can be seen as a recursion tree where the call $C(\bar{v})$ is evaluated as follows: we pick an equation that defines C and an assignment σ that satisfies φ; we evaluate e into v_e,

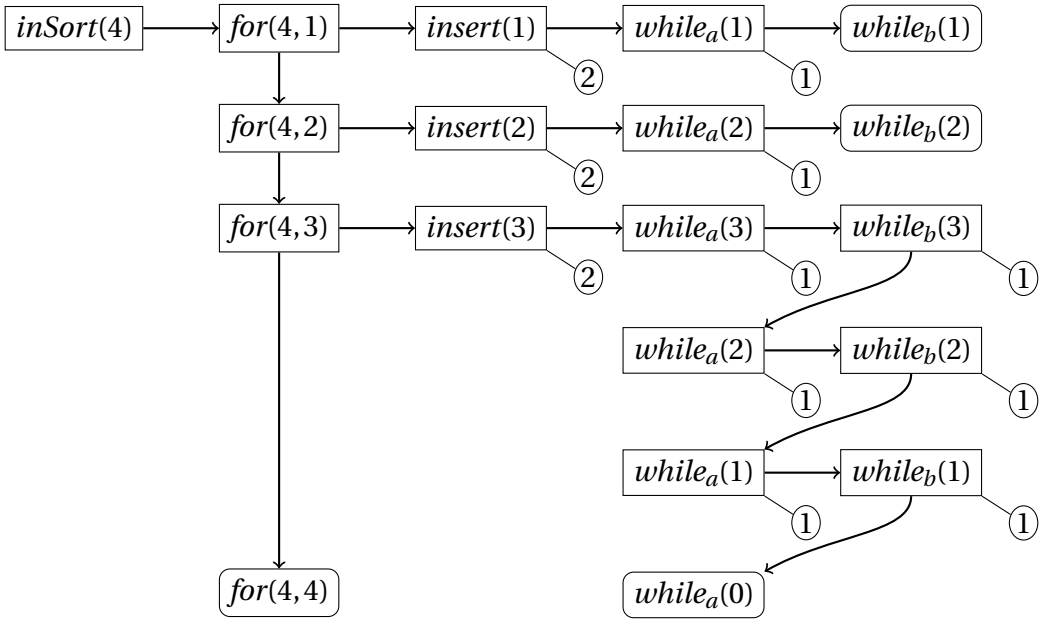


Figure 2.7: Evaluation Tree for a call to the CR *inSort* in Figure 2.6

and recursively call all $C(\sigma(\bar{y}_i))$ simultaneously. Note that an evaluation tree can be infinite. Note also that $C(\bar{v})$ might have several evaluation trees, due to the nondeterminism induced by choosing an equation for C and a satisfying assignment σ for φ. Note that the internal nodes of an evaluation tree correspond to the application of recursive equations, and the leaves correspond to the application of nonrecursive ones.

The set of all evaluation trees for $C(\bar{v})$ is denoted by $\mathit{Trees}(C(\bar{v}))$, and the set of all possible costs is $\mathit{Answers}(C(\bar{v})) = \{\mathsf{Sum}(T) \mid T \in \mathit{Trees}(C(\bar{v}))\}$, where $\mathsf{Sum}(T)$ is the sum of all nodes of T. Then, a function C^+ (resp. C^-) is said to be a UBF (resp. LBF) for the CR C if and only if for any input $\bar{v}$ and $c \in \mathit{Answers}(C(\bar{v}))$, we have $C^+(\bar{v}) \geq c$ (resp. $C^-(\bar{v}) \leq c$).

Example 2.5. An evaluation tree for a call $\mathit{inSort}(4)$, using the CRS of Figure 2.6, is depicted in Figure 2.7. Each rectangle represents a call in the CR with some input values; each circle indicates the cost contributed by the equation that was chosen to resolve that call (as v_e in the explanation above); and directed edges represent calls to other CRs. The cost of this evaluation tree is 11, which coincides with the sum of all circles. ■

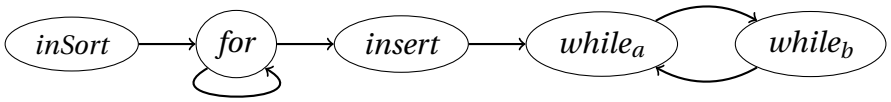
The correctness of the ACR to CRS transformation relies on the idea that every ACR trace has a corresponding evaluation tree with the same cost [10]. Thus, any valid UBF or LBF for the CRS is also valid for the ACR program.

5 Closed-form Upper Bound Functions

In the last step COSTA solves the generated CRS into UBFs or LBFs, one for each CR. For this, it relies on its subsystem PUBS [5, 14]. We first describe how it computes of UBFs, and, at the end of this section, we comment on the case of LBFs, which is less important for presenting our contributions.

The solving procedure of PUBS is designed to handle CRSs in which all recursions are direct, i.e., without mutual calls between different CRs. To handle CRSs that do not meet this requirement, they are first transformed into such form [85, 139]. Essentially, this is done by considering each group of mutually recursive CRs, which coincides with a recursive strongly connected component (SCC for short) in the corresponding call graph, and unfolding all CRs in such a group into one CR (typically the cut-point of the corresponding SCC).

Example 2.6. Consider the CRS of Figure 2.6, and its corresponding call graph:



This graph has two recursive SCCs: one for the CR *for*, which is already in direct recursive form, and one for the CRs $while_a$ and $while_b$, which are mutually recursive. To transform these last CRs into direct recursion, we unfold the definition of $while_b$ into that of $while_a$. This results in a new definition for the CR $while_a$:

$$\begin{array}{ll} while_a(j_0) = 0 & \{j_0 \leq 0\} \\ while_a(j_0) = 1 & \{j_0 \geq 1\} \\ while_a(j_0) = 2 + while_a(j_2) & \{j_0 \geq 1, j_2 = j_0 - 1\} \end{array}$$

The first equation is as the first one of old $while_a$. The second and third equations come from unfolding the call to $while_b$, in the second equation of old $while_a$, using the two equations of $while_b$. The cost expression in the third new equation is the sum of the cost expressions in the old recursive equations of $while_a$ and $while_b$, and its constraint includes those from both equations. ■

In what follows, we may assume that the input CRS is in direct recursive form. The solving procedure of PUBS is an iterative procedure that solves one CR at a time. In particular, in each iteration it solves a stand-alone CR into a corresponding UBF, and then replaces any call to this stand-alone CR by its UBF, thus generating more stand-alone CRs to be solved. This process continues until all CRs are solved. Note that if the CRS is in direct recursion then there is at least one stand-alone CR to start the process, e.g. $while_a$. In what follows we describe how PUBS solves a stand-alone CR into a UBF.

Solving Stand-Alone CRs.

Assume a given stand-alone CR C with n equations, and 1) let bf be the maximum number of recursive calls in any equation of C, i.e., the maximum branching factor of the corresponding evaluation trees; 2) let $e_1, \ldots, e_k$ be the cost expressions of the nonrecursive equations of C; and 3) let $e_{k+1}, \ldots, e_n$ be the cost expressions of the recursive equations of C. PUBS solves C into a UBF by constructing a pessimistic evaluation tree, whose cost is bigger than the cost of any evaluation tree of C, as we explain next. Recall first that, for any evaluation tree, the cost of any internal node is an instance of e_i with $k+1 \leq i \leq n$, and that the cost of any leaf is an instance of e_j with $1 \leq j \leq k$. Now suppose that we have the following:

- a cost expression $h(\bar{x})$ that bounds the height of any evaluation tree. That is, for any input $\bar{v}$ and any evaluation tree $T \in \mathit{Trees}(C(\bar{v}))$, $h(\bar{v})$ is larger than the height of T; and
- a cost expression $\hat{e}_i(\bar{x})$, for each $1 \leq i \leq n$, that bounds the contributions of the i-th equation. Namely, for any input $\bar{v}$ and any evaluation tree $T \in \mathit{Trees}(C(\bar{v}))$, $\hat{e}_i(\bar{v})$ is larger than the cost of any node in T that corresponds to the i-th equation.

Now we construct a pessimistic evaluation tree T (parametric in the input $\bar{x}$) as follows:

1. T is a complete tree, with branching factor bf and height $h(\bar{x})$;
2. each leaf of T has cost $\max(\hat{e}_1(\bar{x}), \ldots, \hat{e}_k(\bar{x}))$; and
3. each internal node of T has cost $\max(\hat{e}_{k+1}(\bar{x}), \ldots, \hat{e}_n(\bar{x}))$.

The number of leaves $\mathscr{L}$ in T is $bf^{h(\bar{x})}$, and the number of internal nodes $\mathscr{N}$ is $\frac{bf^{h(\bar{x})}-1}{bf-1}$ if $bf > 1$ and $h(\bar{x})$ otherwise. The cost of T is

$$C^+(\bar{x}) = \mathscr{L} * \max(\hat{e}_{k+1}(\bar{x}), \ldots, \hat{e}_n(\bar{x})) + \mathscr{N} * \max(\hat{e}_1(\bar{x}), \ldots, \hat{e}_k(\bar{x}))$$

which is a valid UBF for C as well.

For automatically inferring $h(\bar{x})$, PUBS relies on the use of linear ranking functions [26], which are intensively used to bound the number of iterations of loops. It is easy to see that the height of a tree corresponds to the number of consecutive recursive call in C, which is a form of a loop. For automatically inferring $\hat{e}_i(\bar{x})$, PUBS relies on a **maximisation** procedure that is based on invariants generation. The details are not important to explain our contributions, the interested reader may find these details in [5].

Example 2.7. Let us demonstrate the different steps of PUBS on the CRS of Figure 2.6, assuming that it has been transformed already to direct recursive form as in Example 2.6:

- We start by solving CR $while_a$ of Example 2.6, which is the only stand-alone CR. The height of its evaluation trees is bounded by $h(j_0) = \mathsf{nat}(j_0)$, and the cost expressions of the nonrecursive and recursive equations are (trivially) maximised to 1 and 2 respectively. Therefore, we get the UBF ${while_a}^+(j_0) = 2 * \mathsf{nat}(j_0) + 1$.
- Substituting ${while_a}^+(j_0)$ in the nonrecursive CR *insert* results in the stand-alone CR defined with this only equation:

$$insert(i_0) = 2 + 2 * \mathsf{nat}(i_0) + 1 \quad \{\}$$

  which is trivially solved to the UBF $insert^+(i_0) = 2 * \mathsf{nat}(i_0) + 3$.
- Substituting $insert^+(i_0)$ in CR *for* results in the following stand-alone CR:

$$\begin{array}{ll} for(l_0, i_0) = 0 & \{i_0 \geq l_0\} \\ for(l_0, i_0) = 2 * \mathsf{nat}(i_0) + 3 + for(l_0, i_2) & \{i_0 + 1 \leq l_0, i_2 = i_0 + 1\} \end{array}$$

  The height of its evaluation trees is bounded by $h(l_0, i_0) = \mathsf{nat}(l_0 - i_0)$. The cost expression $2 * \mathsf{nat}(i_0) + 3$ is maximised to $\hat{e}(l_0, i_0) = 2 * \mathsf{nat}(l_0 - 1) + 3$ because the value of i_0 can be at most $l_0 - 1$, where l_0 refers to the initial value with which *for* is called, not to the parameter l_0. Therefore, we get the UBF $for^+(l_0, i_0) = \mathsf{nat}(l_0 - i_0) * (2 * \mathsf{nat}(l_0 - 1) + 3)$.
- Substituting this UBF in *inSort* results in the following stand-alone CR:

$$inSort(l_0) = \quad \mathsf{nat}(l_0 - i_2) * (2 * \mathsf{nat}(l_0 - 1) + 3) \quad \{i_2 = 1\}$$

  The height of any evaluation tree is 0 since it is nonrecursive, and the maximisation of $\mathsf{nat}(l_0 - i_2) * (2 * \mathsf{nat}(l_0 - 1) + 3)$ results in $\mathsf{nat}(l_0 - 1) * (2 * \mathsf{nat}(l_0 - 1) + 3)$ since $i_2 = 1$. Therefore, for this CR we get the UBF $inSort^+(l_0) = \mathsf{nat}(l_0 - 1) * (2 * \mathsf{nat}(l_0 - 1) + 3)$.

Note that the UBFs are asymptotically tight, in the sense that the worst-case cost of `inSort` is quadratic in the parameter `last`. However, they are not functionally tight, because the overapproximation is too coarse. ■

Inference of LBFs in PUBS is done as described in [14]. It relies on approximating the behaviour of the input CRS by a corresponding system of RRs, and then using computer algebra systems to solve it. For example, it would infer the LBF $inSort^-(l_0) = 3 * \mathsf{nat}(l_0 - 1)$, which corresponds to the case in which the input array is sorted, and thus the body of `while` loop is never executed.

3 | Asymptotic Closed-Form Bounds

In this Chapter we describe a method to transform a bound function into a reduced asymptotic form, and how to use this transformation to build an **asymptotic cost analyser**, which directly computes bounds in asymptotic form. This contribution has been published in the following article:

> ELVIRA ALBERT, DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, SAMIR GENAIM, AND GERMAN PUEBLA. Asymptotic Resource Usage Bounds. In Zhenjiang Hu, editor of the Proceedings of *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 5904 of *Lecture Notes in Computer Science*, pages 294–310. Springer, December 2009

which is attached to this dissertation at Page 195.

Overview

Asymptotic notations are used to succinctly describe how the cost of a program scales with the size of its input. The key observation behind them is that:

> "Although we can sometimes determine the exact running time of an algorithm [...], the extra precision is not usually worth the effort of computing it. For large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself." [45, Chapter 3]

Asymptotic notations ignore those constants and lower terms, and focus instead on the proportionality, up to some scalar and for large input, between the worst-case (or best-case) cost of a program and a function of (the sizes of) the input of the program. This proportionality or **asymptotic equivalence** is a condition so lax that the worst-case (or best-case) cost can be asymptotically equivalent to infinitely many functions; yet, among them, there is a simple one like 1, n^2, $n\log n$, or 2^n, which we call the **asymptotic form**. A bound functions on the program's cost in asymptotic form is called an **asymptotic closed-form bound**[1], i.e., an asymptotic LBF (ALBF for short) or an asymptotic UBF (AUBF for short).

For some applications, we need a bound with all the *multiplicative constants and lower-order terms*, i.e., non-asymptotic. For instance, to split tasks

[1] Also called the asymptotic complexity of the program.

among processors [95], to decide if and where to migrate a process [55], to avoid running out of battery [86], or to avoid a heap overflow [13], we cannot rely on information that tells us that the cost is eventually linear by some constant. In general, we need non-asymptotic bounds for the verification, compilation, and optimizing execution of programs.

In the above scenarios, we usually consider programs whose development has terminated already. However, we should not wait until the end of the development process to (automatically) check whether a program meets its (cost) specifications or not [91], we should try to apply this check frequently in order to detect performance bugs during the development process as early as possible. This can be done, for instance, by running the analysis and providing the programmers with the inferred bounds; also, they may write in the code an assertion on the cost of a method, and run the analyser to verify it. In such cases, using asymptotic bounds have clear advantages over non-asymptotic ones:

- As it is more concise, an asymptotic bound is easier to read than a non-asymptotic one. Thus, using asymptotic bounds improves the usability for the programmers of the analyser.
- The programmers' main concern before writing the program is its scalability, about which they may have an idea clear enough to write it down as an assertion of an asymptotic bound. Thus, these assertions are easier to figure out. Instead, if they were asked to write an assertion of a non-asymptotic bound, they might not know what coefficients and minor expressions to put in it, because such information depends on implementation details that they can not predict before writing the program.
- The non-asymptotic bound for a program under development is volatile, because the *multiplicative constants and lower-order terms* can change due to any change in the program, in the compiler, or in the libraries. As these things happen often during the development, if non-asymptotic bounds were used then there would be too many and too frequent notifications. Asymptotic bounds are more stable, since only major changes or improvements have an impact on the program's asymptotic complexity.
- Once the program is written, the programmers have to improve its performance and scalability. A guiding principle for this task is to focus on the program's bottlenecks, that is the most frequent or expensive operations:

  > "Programmers waste enormous amounts of time thinking about [...] the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. [...] A good programmer [...]

> will be wise to look carefully at the critical code; but only after that code has been identified." [87, page 268]

We can use an asymptotic cost analyser to identify that critical code –the bottlenecks– at which we should direct the optimisation effort.

- Since asymptotic bounds are smaller, an asymptotic cost analyser can be faster and more scalable than a non-asymptotic one. Having a faster analyser is especially relevant for integrating it in an interactive tool that compiles and analyses the code just as it is written.

Thus, an asymptotic cost analyser can be an easy to use tool for analysing and improving the performance of the program.

In Section 3.1, we recall the relevant properties of the asymptotic notations, and we adapt these notations to the cost expressions of PUBS. In Section 3.2 we describe our asymptotic transformation, which is an algorithm to transform a cost expression into another one that is asymptotically equivalent. In Section 3.3 we describe our asymptotic comparison, which is an algorithm to prove that a cost expression is asymptotically greater than another one. In Section 3.4 we discuss how to use this transformation to build a cost analyser that directly generates asymptotic bounds. In Section 3.5 we discuss an experimental evaluation of the techniques. In Section 3.6 we overview related work, and in Section 3.7 we describe some further details that can be found in the article.

1 Asymptotic Notations for Cost Expressions

In this Section we recall the critical properties of the asymptotic notations for univariate functions, and we adapt these notations to the cost expressions of PUBS (see Section 2.4).

The asymptotic notations big Omicron (O) and big Theta (Θ), are commonly defined as follows. If g is a univariate function, that is a function in $\mathbb{N} \mapsto \mathbb{R}^+$, then $O(g)$ and $\Theta(g)$ denote sets of univariate functions. In particular, $f \in O(g)$ if, for large values of the input n, $f(n) \leq c_u g(n)$ for some constant $c_u > 0$. Similarly, $f \in \Theta(g)$ if $c_l g(n) \leq f(n) \leq c_u g(n)$ for large values of the input n and for some coefficients $c_l, c_u > 0$.

These notations are frequently used to compare functions, so we can read $f \in O(g)$ as "f is asymptotically smaller than g", or "g is an asymptotic upper bound of f"; and we can read $f \in \Theta(g)$ as "f is asymptotically equivalent to g", or "g is an asymptotically tight bound of f". In this sense, $f \in \Theta(g)$ is an equivalence relation and $f \in O(g)$ is an order relation [2], which have the following properties:

[2] Some authors [61, §4.1] [111, §9.1] use $f \preceq g$ and $f \approx g$ as shorthands of $f \in O(g)$ and $f \in \Theta(g)$.

1. The exact asymptotic order Θ of a function does not change if we multiply it by a constant d, formally $\Theta(d * f) = \Theta(f)$.
2. The product of functions is monotonic with respect to the order relation defined by O. Formally, if $f' \in O(f)$ and $g' \in O(g)$ then $f' * g' \in O(f * g)$.
3. The exact asymptotic order of a sum of two terms is that of its dominating term, formally if $f \in O(g)$ then $\Theta(f + g) = \Theta(g)$.

These properties are essential for automatically transforming a function to its asymptotic form, or for comparing two functions, because we can use them to deduce some easy to implement rules for comparing two expressions. For instance, we get the rule to compare the complexity of two polynomials by comparing their degrees, or we can compare two exponential expressions by comparing their bases[3]. Due to these properties, Θ of a simple function (like n or n^2) may contain more complex functions.

Example 3.1. Let $f(n) = 3n^2 + n\log_3 n + 5n$, we can omit the multiplicative coefficients 3 and 5 due to the first property. Then, since both n and $n\log_3 n$ are in $O(n^2)$, using the third property we get $f \in \Theta(n^2)$. ■

The classical definition refers to univariate functions, i.e., functions with one natural argument. However, a bound can be a multivariate function, that is, a function of many natural variables, either because the program has many input variables [66, 65, 84], or because the bound function is defined over several size measures for each input [73, 62, 6]. Since there is no standard definitions for the multivariate functions nor a proof of their properties, many people simply apply the asymptotic notations to multivariate functions as if they had the same properties as the univariate ones. However, Howell [77] shows that some properties of the asymptotic notations, which are crucial for building an asymptotic transformation and comparison method, do not hold when the notations are generalised to all multivariate functions.

Fortunately, from the work of Howell we can deduce two conditions to define the asymptotic notations for multivariate functions, in such a way that the crucial properties are preserved:

- The asymptotic notations must be restricted to multivariate functions that are monotonic (non-decreasing) on all their inputs; and
- The asymptotic notations must refer to the relation between the functions for input vectors in which all components are large.

For this reason, we define the asymptotic notations for multivariate functions [2, Definition 2] by generalising the classical definitions for univariate functions. This generalisation is done by replacing the symbol n of an input variable by

[3] Formally, for $0 \le a \le b$, we have that $n^a \in O(n^b)$, and for $1 < a \le b$ we get that $a^n \in O(b^n)$.

a symbol $\bar{n}$ of an input vector. For instance, for any multivariate function g in $\mathbb{N}^m \mapsto \mathbb{R}^+$, $O(g)$ denotes the set of functions f in $\mathbb{N}^m \mapsto \mathbb{R}^+$ for which there is a real $c_u > 0$ such that $f(\bar{n}) \leq c_u g(\bar{n})$, for large input vectors. Here, large input vector means one in which all components are large.

Example 3.2. Consider the multivariate function $f(A,B) = A*(2*B+3)$. Using the above properties we can deduce that $2*B+3 \in \Theta(B)$, and therefore $f(A,B) \in \Theta(A*B)$. ■

Since cost expressions are functions over integer variables, before we use the above definition of asymptotic notations, we need first to transform them into (or approximate them by) ones over natural variables, that moreover are monotonic on each component. For that, we need to identify some elements of cost expression that we can abstract to natural variables, such that the resulting expression is monotonic on those elements: we chose the nat subexpressions as those elements (see Section 2.4). The next example explains this choice.

Example 3.3. Consider the cost expression $\mathsf{nat}(l_0 - i_0) * (2 * \mathsf{nat}(l_0 - 1) + 3)$ of Example 2.7. Since it is monotonic on its nat components, we can abstract it to the function $A*(2*B+3)$, where A and B are natural variables that abstract $\mathsf{nat}(l_0 - i_0)$ and $\mathsf{nat}(l_0 - 1)$ respectively. We cannot abstract this cost expression by mapping l_0 and i_0 to natural variables since the cost expression is not monotonic on i_0. ■

We define the asymptotic notations for cost expressions e as a multivariate function f_e, where every natural argument of f_e corresponds to a nat subexpression of e. This abstraction should map different nat subexpressions to different variables of f_e, but the same natural variable should be used for nat subexpressions that appear more than once. We call f_e the nat-free abstraction of e, and its variables the nat variables.

Example 3.4. In Example 3.3, for $e \equiv \mathsf{nat}(l_0 - i_0) * (2 * \mathsf{nat}(l_0 - 1) + 3)$, we used the nat-free abstraction $f_e \equiv A*(2*B*+3)$. ■

The asymptotic relation between cost expressions is defined just as the relation between their nat-free abstractions. Namely, if f_1 and f_2 are the nat-free abstractions of e_1 and e_2 respectively, then $e_1 \in O(e_2)$ if $f_1 \in O(f_2)$.

Using the nat-free abstraction to define the asymptotic notations carries a basic intuition in complexity analysis [15, 84, 65]. The asymptotic complexity of the program is related directly to the number of iterations that each loop performs. As we have seen in Section 2.5, the number of iterations of a loop corresponds to the height of an evaluation tree, which is bounded by a cost expression $\mathsf{nat}(l)$ where l is a linear ranking function. Thus, the nat expressions

are the main components that affect the cost of the program, so we may define the complexity of the program as the one in which the nat subexpressions take large values.

2 Asymptotic Transformation

In general, an asymptotic transformation [126, 2] is a procedure that takes a cost expression e and returns a simpler and asymptotically equivalent expression e'. Such a transformation consists of a series of steps to remove redundant operands and operations: if a and b are two expressions and $\diamond$ is an operation, we say that b is a redundant expression if $a \diamond b \in \Theta(a)$, so b does not change the asymptotic complexity of the expression. In our case, our asymptotic transformation takes as input a bound given as a pair $\{e, \varphi\}$, where e is a cost expression and φ is a a conjunction of linear constraints that restricts the values of the variables of e. In the following we say that φ is a context constraint. Our asymptotic transformation works in three steps:

1. **Computing the** nat**-free abstraction** f_e as described in the previous section. However, to avoid generating too many nat-variables, we first modify e such that proportional nat sub-expressions are replaced by a common one. Thus, they will be abstracted to the same nat-variable.
2. Transforming the nat-free expression f_e into an **asymptotic normal form** by performing the following operations: *1*) replace every max operation $\max(e_1, e_2)$ by $e_1 + e_2$; *2*) remove all multiplicative constants; *3*) replace each exponential cost expression $b^A - 1$ by b^A; *4*) replace each logarithmic cost expression $\log_b(A+1)$ by $\log A$; and *5*) rewrite the expression as a sum of products of basic cost expressions (Section 2.4) .
3. **Remove redundant terms** from the normalised cost expression. Those addends in the sum that are asymptotically smaller than others, and thus do not modify the asymptotic order of the sum.

The first two steps are easy to implement, since they are just two syntactic transformations. For the last step it is necessary to use an asymptotic comparison that we describe in the next section.

Example 3.5. Consider the cost expression $e = \mathsf{nat}(l_0) * (2 * \mathsf{nat}(l_0 - 1) + 3)$ of Example 2.7. Our transformation proceeds as follows: (*1*) $\mathsf{nat}(l_0 - 1)$ is replaced by $\mathsf{nat}(l_0)$ and then the nat-free abstraction results in $f_e = A * (2 * A + 3)$; (*2*) the constants in f_e are removed, and then f_e is transformed into the normal form $A^2 + A$; (*3*) the redundant term A is removed to obtain A^2 . To get A^2 in terms of the original nat expressions, we simply undo the nat-free abstraction which results in $\mathsf{nat}(l_0)^2$. ■

3 Asymptotic Comparison

Our asymptotic comparison procedure takes two nat-free expressions e_1 and e_2, together with a context constraint φ, and tries to prove that $e_1 \in O(e_2)$. It is based on the notions of asymptotic subsumption and asymptotic weight:

- **Asymptotic subsumption**. If A and B are nat-variables that correspond to $\mathsf{nat}(l_1)$ and $\mathsf{nat}(l_2)$ respectively, we say that A subsumes B (modulo φ), written as $A \succcurlyeq B$, if φ implies that $\mathsf{nat}(l_1) \in O(\mathsf{nat}(l_2))$.
- **Asymptotic weight**. It is the key measure to compare the growth of each expression, such as the base of the exponential factors, or the degree of the polynomials or poly-logarithmic[4] factors. Comparing the weights of expressions is done by first comparing the exponential base, then the degree of the polynomials and finally the degree of the poly-logarithms.

These notions provide a straightforward way to compare two cost expressions e_1 and e_2, modulo a context constraint φ:

(R1) To prove that $P \in O(b)$, where b is a basic cost expression and P is a product of basic expressions, we have to check 1) that the nat-variable of b subsumes (modulo φ) every nat-variable of P; and 2) that b has a greater asymptotic weight than P.

(R2) To prove that $P \in O(Q)$, where P and Q are products of basic cost expressions, we try to factorise P into k sub-products $P = p_1 * p_2 * \ldots * p_k$, such that there exists k different factors b_i in Q verifying $p_i \in O(b_i)$.

(R3) To prove that $S \in O(T)$, where S and T are sums of products of basic cost expressions, we only have to find, for each addend a in S, an addend a' in T for which $a \in O(a')$.

Let us see an example of the above comparison rules.

Example 3.6. Let us compare the nat-free expressions $e_1 \equiv 2^B C^2 + A^3 D + D^2 A$ and $e_2 \equiv B^7 C + A^2 \log^2 B + B^2 C^2 + D^2 \log C$, where $A = \mathsf{nat}(x+y)$, $B = \mathsf{nat}(x)$, $C = \mathsf{nat}(y)$, and $D = \mathsf{nat}(z)$. Assuming the context constraint $\varphi \equiv \{x \geq 0, y \geq 0, z \geq y\}$, we can deduce the subsumption relations $A \succcurlyeq B, A \succcurlyeq C, D \succcurlyeq C$. Now we can build a proof of $e_2 \in O(e_1)$ as follows:

- Using (R1) we deduce the following relations:

$$B^7 \in O\left(2^B\right) \quad A^2 \log^2 B \in O\left(A^3\right) \quad B^2 C \in O\left(A^3\right) \quad C \in O\left(C^2\right)$$
$$C \in O(D) \qquad \log C \in O(A) \qquad D^2 \in O\left(D^2\right)$$

  In the case of $A^2 \log^2 B \in O(A^3)$, we have that A subsumes both A and B, and the asymptotic weight of A^3 is greater than that of $A^2 \log^2 B$.

[4] A poly-logarithm is an expression like $(\log A)^2$.

- Using (R2) we deduce the following relations:

$$B^7C \in O(2^B C^2) \qquad B^2C^2 \in O(A^3D)$$
$$A^2\log^2 B \in O(A^3D) \qquad D^2\log C \in O(D^2A)$$

  In the case of $A^2\log^2 B \in O(A^3D)$, we only use one sub-product and one factor since $A^2\log^2 B \in O(A^3)$. In $B^2C^2 \in O(A^3D)$, we factorise B^2C^2 into $B^2C * C$ and we use the relations $B^2C \in O(A^3)$ and $C \in O(D)$.
- Using (R3) we deduce that

$$B^7C + A^2\log^2 B + B^2C^2 + D^2\log C \in O\left(2^B C^2 + A^3D + D^2A\right)$$

  because for every addend on the left there is one addend on the right that is asymptotically bigger (as shown in the previous step).

Thus, we conclude that $e_2 \in O(e_1)$. ■

The comparison procedure above is correct, however it is not complete. I.e., there exists cases in which $e_1 \in O(e_2)$ but the procedure is unable to prove it.

Example 3.7. For $e_1 \equiv A^2 + B^2$ and $e_2 \equiv AB$, it holds that $e_1 \in O(e_2)$, but our automatic comparison fails to prove it. ■

4 Asymptotic Cost Analysis

In this section we discuss how to build an asymptotic cost analyser that directly infers asymptotic bounds. A straightforward solution is to feed the output of an existing non-asymptotic cost analyser [73, 66, 57, 90, 130] to the input of the asymptotic transformation.

Example 3.8. If we apply the asymptotic transformation to the UBFs of Example 2.7, we get the following asymptotic bounds:

$$\mathit{inSort}(l_0) = \mathsf{nat}(l_0)^2 \qquad \mathit{insert}(i_0) = \mathsf{nat}(i_0)$$
$$\mathit{for}(l_0, i_0) = \mathsf{nat}(l_0 - i_0) * \mathsf{nat}(l_0) \qquad \mathit{while}_a(j_0) = \mathsf{nat}(j_0)$$

which are, precisely, what we would obtain by manually analysing the asymptotic cost of the methods and loops of the `inSort` program in Figure 2.2. ■

This solution is correct, but it is inefficient because the cost analyser generates detailed information that is thrown away by the asymptotic transformation. To avoid generating these details, we have developed an **asymptotic CRS**

solver that directly generates asymptotic bounds. The advantage of this approach is its efficiency, however, it can be applied only to cost analysers that are based on generating CRSs and then solving then into corresponding bounds.

Our solver follows the same phases as PUBS (see Section 2.5), however, it also applies the asymptotic transformation immediately after a cost expression is generated. In particular, we apply it at the following steps of PUBS:

- When transforming a CRS to direct recursive form, PUBS generates cost expressions that are collected from the unfolded equations. We apply the asymptotic transformation to such expressions.
- When transforming each CR in the CRS to a stand-alone form, PUBS replaces each external call by the bound computed for the called CR, and adds the bounds to the cost expression of the corresponding equation. We apply the asymptotic transformation to such expressions.
- When solving a standalone CR into a non-asymptotic bound, before storing this bound function we also apply the asymptotic transformation.

We do not change the way in which PUBS computes a ranking function, maximises cost expressions, and solves a standalone CR to bound functions.

Example 3.9. Let us apply the asymptotic resolution to solve the CRS of Figure 2.6. Transforming this CRS to direct recursion is done as in Example 2.6, except that when generating the third equation of $while_a$, the asymptotic transformation replaces the constant 2 with 1. The cost expressions of the other equations are already in asymptotic form. The next step is to solve the CRS, one CR at a time, similarly to what we have done in Example 2.7:

- The UBF of the CR $while_a(j_0)$ is $\mathsf{nat}(j_0)+1$. The asymptotic transformation reduces it to the AUBF ${while_a}^+(j_0) = \mathsf{nat}(j_0)$.
- Substituting ${while_a}^+(j_0)$ in the CR *insert*, results in a stand-alone CR with a single equation in which the cost expression is $1+\mathsf{nat}(i_0)$, which is then reduced by the asymptotic transformation to $\mathsf{nat}(i_0)$. Then, we get the UBF $insert^+(i_0) = \mathsf{nat}(i_0)$.
- Substituting $insert^+(i_0)$ in the CR *for*, results in a stand-alone CR in which the cost expression of the recursive equation is $\mathsf{nat}(i_0)$, which is already in asymptotic form. Solving this CR results in the UBF $\mathsf{nat}(l_0-i_0)*\mathsf{nat}(l_0-1)$, which is then reduced by the asymptotic transformation to $for^+(l_0,i_0) = \mathsf{nat}(l_0-i_0)*\mathsf{nat}(l_0)$.
- Substituting $for^+(l_0,i_0)$ in the CR *inSort*, and then solving it, results in the UBF $\mathsf{nat}(l_0-1)*\mathsf{nat}(l_0)$, which is then reduced by the asymptotic transformation to $inSort^+(l_0) = \mathsf{nat}(l_0)^2$.

Note that the UBFs computed above are as those of Example 3.8. ■

5 Experimental Evaluation

In this section we discuss an experimental evaluation of our techniques, which mainly compares our techniques to those of PUBS for inferring non-asymptotic bounds. Note that there are no other techniques available that can compute asymptotic bounds.

Evaluation of the Asymptotic Transformation. We implemented our transformation as a back-end of COSTA. To experimentally evaluate it, we applied it to transform to asymptotic form the UBFs on memory consumption obtained by [13, §7]. For all benchmarks, the transformation obtained an accurate and minimal asymptotic form, and it took a negligible time. This shows how our transformation enables the transfer of existing work and software on computing non-asymptotic closed-form bounds into generating asymptotic closed-form bounds. This technique is applicable both to UBFs and LBFs, for any cost model and for any size measure.

Evaluation of the Asymptotic Resolution. In a second set of experiments, we studied the scalability of our approach, that is, how the size of the cost expressions and the time required to compute it increase when solving larger CRs. We have used the same set of benchmarks that were used in [5, §10.2] to study the scalability of PUBS. For each benchmark, we computed both a UBFs (using PUBS) and an AUBFs, and observed the following: (1) the time required to compute a UBF grows significantly with the size of the CRs, while the time to compute an AUBF remains small; (2) the size of each UBF is significantly larger than its corresponding AUBF; (3) the ratio between the size of the computed bounds and the number of equations grows faster for non-asymptotic UBFs than for asymptotic ones; and (4) for some of the biggest benchmarks, it was not possible to compute a non-asymptotic UBF in a reasonable time, but it was possible to compute the asymptotic form. These observations demonstrate that our approach is scalable.

6 Related Work

Asymptotic notations. Knuth [88] gave the formal definition of the asymptotic notations for univariate functions. Some authors [45, §3.1],[111, §9.2] use this definition because it is easy to translate to a logical formula from which to prove some basic properties. Others [118, §1.2], [61, §4.1.1], use another equivalent definition of O and Θ, based on the limit of the quotient $f(n)/g(n)$ when n tends to infinity. This is a more intuitive definition, as it better conveys the

idea that we study the proportionality between functions for arbitrarily large values of the input. Our work is centered on the big-O and big-Theta notations. However, it can be extended to big-Omega Ω, little-o $o(f)$ and little-omega $\omega(f)$ notations.

A fundamental part of our work was to extend these notations to multivariate natural functions, and, at the same time, preserve the corresponding algebraic properties. We solved this issue by restricting the definition to functions that are monotonic on all input components as shown by Howell [77].

Asymptotic Transformation and Comparison. Stoutemyer [126] presents another method for the asymptotic transformation and comparison of functions. His procedure is built on the MACSYMA computer algebra system, and uses some of its advanced features. For instance, to compute limits of function quotients or Taylor series that cover a wide set of expressions. On the other hand, our method can handle context information (in the form of linear constraints). Our comparison method has some similarities with the method in PUBS for comparing non-asymptotic cost expressions [4].

Asymptotic Cost Analysis. Early works on static cost analysis [41, 43] distinguish between macro and micro analysis of programs, where the first one focuses on a dominant operation and the latter considers all operations. Under this view, an asymptotic bound falls into the category of macro analysis. In [101], an automatic asymptotic cost analysis for Horn clauses was presented, which has to take into account the sparsity of the relations in the logic program. The ACE system [96] performs an asymptotic cost analysis, which is based on program transformation, for programs in a first-order functional language. Complexity analyses have also been developed for other languages, such as simple loop programs [84], or term rewriting systems [102], and through the later for logic programming languages [59].

7 Further Reading

In this Chapter we have discussed the main contributions of [2], which are 1) the extension of the asymptotic notations to cost expressions; 2) an asymptotic transformation and comparison for cost expressions; and 3) an asymptotic CRS solver. We have illustrated the transformation and analysis with a simple example. The article contains the definitions of the asymptotic notations for cost expressions, and the definitions of the asymptotic transformation and the asymptotic comparison for cost expressions. It also provides correctness statements and their corresponding proofs.

4 | Nonlinear Operations

In this Chapter we describe a value analysis that is able to handle nonlinear integer arithmetic operations. When used within COSTA, it allows inferring precise asymptotic bound functions for programs on which COSTA failed or inferred imprecise ones before. This contribution has been published in the following article:

> DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Handling Non-linear Operations in the Value Analysis of COSTA. *Electronic Notes in Theoretical Computer Science*, 279(1):3–17, 2011. Proceedings of Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)

which is attached to this dissertation at Page 217.

Overview

Our major goal in this dissertation is to improve the precision of COSTA, so that it computes precise bound functions for a wider class of programs. In this Chapter we focus on those components that abstract RBR instructions to linear constraints, infer postconditions for the different ACR procedures by means of linear constraints, and incorporate all these linear constraints in the corresponding CRS. Our goal is to improve the accuracy of such components so that they compute tighter approximations, or equivalently stronger constraints. The strength of these constraints has a direct impact on the precision of the inferred bounds: if they are too weak, then the CRS might include spurious evaluations that correspond to no program execution.

The constraints in the CRS approximate the values of the program variables at corresponding program points, and also define relations between the values of these variables. As we have seen in Chapter 2, these constraints come from either the abstract compilation or the value analysis. The value analysis of COSTA is based on the theory of Abstract Interpretation [48, 46]. This theory starts from the observation that the exact set of value relations is not computable. Instead, an abstract interpretation uses an abstract domain, which is a syntactic class of formulas [34, §12.1.4], that (typically) restricts the shape of the relations that can be used to either approximate the effect of executing a sequence of instructions, or to represent the inferred value relations.

Both the abstract compilation and the value analysis are based on the use of conjunctions of linear constraints which are enough, in practice, for efficiently

```
1   static int clog(int x, int b){
2       if (b > 1) {
3           int y = 1 ;
4           int z = 0 ;
5           while ( y < x ){
6               y = y * b ;    ★
7               z = z + 1 ;
8           } ;
9           return z ;
10      } else  return 0 ;
11  }
```

Figure 4.1: JAVA code for the `clog` example. We mark the product with ★.

and precisely handling a wide class of programs. However, they are not enough for modeling the effect of nonlinear operations, like for instance the product $z = x * y$, whose semantics cannot be precisely modeled with a conjunction of linear constraints. This clearly affects the precision of the inferred bounds if the cost depends on such instructions.

The main problem we are dealing with, in this Chapter, is the imprecision in the abstract compilation and value analysis caused by such nonlinear arithmetic instructions. To solve this problem, we notice that although we cannot model these operations using a *single* conjunction of linear constraints, we can do so if we use a finite *disjunction* of such conjunctions by splitting the semantics of each nonlinear operation into several cases. Moreover, we can encode such disjunctions as auxiliary procedures in the ACR program, and thus, we can still use a scalable value analysis, and at the same time benefit from the disjunctive information to infer more precise value relations.

In Section 4.1 we explain the inner workings of our approach; In Section 4.2 we discuss an experimental evaluation of the techniques. in Section 4.3 we overview related work; and in Section 4.4 we describe some further details that can be found in the article.

1 Handling a Nonlinear Operation

The abstract compilation of COSTA abstracts nonlinear operations to *true*, which represents the space of all program states. This results in a significant loss of precision that leads to inferring asymptotically imprecise bounds, as demonstrated in the following example.

$clog(\langle x, b\rangle, \langle r\rangle) \leftarrow$
  $if_c(\langle x, b\rangle, \langle r\rangle)$.
$if_c(\langle x, b\rangle, \langle r\rangle) \leftarrow$
  $\{\neg b > 1\}$,
  $r := 0$.
$if_c(\langle x, b\rangle, \langle r\rangle) \leftarrow$
  $\{b > 1\}$,
  $y := 1$,
  $z := 0$,
  $while_c(\langle x, b, y, z\rangle, \langle z\rangle)$,
  $r := z$.
$while_c(\langle x, b, y, z\rangle, \langle z\rangle) \leftarrow$
  $\{\neg y < x\}$.
$while_c(\langle x, b, y, z\rangle, \langle z\rangle) \leftarrow$
  $\{y < x\}$,
  $y := y * b$, ★
  $z := z + 1$,
  $while_c(\langle x, b, y, z\rangle, \langle z\rangle)$.

$clog(\langle x_0, b_0\rangle, \langle\rangle) \leftarrow$
  $if_c(\langle x_0, b_0\rangle, \langle\rangle)$.
$if_c(\langle x_0, b_0\rangle, \langle\rangle) \leftarrow$
  $b_0 \leq 1$.

$if_c(\langle x_0, b_0\rangle, \langle\rangle) \leftarrow$
  $b_0 \geq 2$,
  $y_0 = 1$,
  $while_c(\langle x_0, b_0, y_0\rangle, \langle\rangle)$.

$while_c(\langle x_0, b_0, y_0\rangle, \langle\rangle) \leftarrow$
  $y_0 \geq x_0$.
$while_c(\langle x_0, b_0, y_0\rangle, \langle\rangle) \leftarrow$
  $y_0 + 1 \leq x_0$,
  $true$, ★
  acquire(1),
  $while_c(\langle x_0, b_0, y_1\rangle, \langle\rangle)$.

Figure 4.2: The RBR (left) and the ACR (right) obtained by COSTA for the `clog` example. We mark the nonlinear instruction with ★.

Example 4.1. Figure 4.1 depicts a Java method `clog`, which takes as input two positive numbers `x` and `b`, and returns the value of $\lceil \log_b(x) \rceil$. We are interested in analysing the cost of this program with respect to a cost model that counts, for example, the number of visits to the loop body. Note that the precise asymptotic UBF in such case is in $\Theta(\log(x))$. The corresponding RBR and ACR programs, generated by COSTA, are depicted in Figure 4.2. Note that the RBR variables r and z were removed in the ACR because they do not affect the cost. In this example, the loss of crucial information happens in the abstract compilation step. In the JAVA method (resp. RBR program), the instruction `y = y * b` (resp. $y := y * b$) updates `y` (resp. y) to hold the value of `y * b` (resp. $y * b$). However, the constraint *true* in the ACR program means that variable y_1 may take any value, and this causes the cost of this ACR program to be unbounded. ■

In order to solve the above precision problem, and infer a precise UBF for method `clog`, we need to handle the product operation more precisely. That is, we want to improve the abstract compilation and value analysis to obtain a more precise value relation for that instruction. For this, we could use abstract domains that are able to track nonlinear relations [115, 28, 63]. However, operations in these domains are computationally expensive, which would render

the value analysis impractical in most cases.

Our solution is based on the following observation: although the instruction is not linear, it can be approximated by a linear relation under certain contexts. That is, if a linear constraint over the values of the operands (the context) holds, then an inequality over the result, or a relation between its input and output variables, holds as well.

Example 4.2. Consider the instruction $y := y * b$ in the RBR program of Figure 4.2. If we have no information about the values of y and b, then the constraint *true* is the only correct way of abstracting the semantics of $y := y * b$ with a conjunction of linear constraints. Instead, if the constraint $y \geq 1 \wedge b \geq 2$ is known to hold before the instruction, then we can use this context information to refine the abstraction of $y := y * b$ to $y_1 \geq 2 * y_0$, where y_0 and y_1 represent the values of y before and after the instruction, respectively. ■

The essence of our solution is to use context-sensitive information to refine the abstraction of nonlinear operations. Namely, we abstract each such instruction to a disjunction of cases, where each case specifies a possible scenario using a conjunction of linear inequalities. Note that for this abstraction to be correct, these cases must cover the whole input domain. The actual challenge here is how to represent these disjunctions and at the same time keep the value analysis practical.

An immediate solution would be to use disjunctive abstract domains, like powerset of polyhedra. However, these domains usually come with a performance overhead, which renders the analysis impractical in many cases. Moreover, we note that this disjunctive information is not required globally, but only locally when analysing the effect of nonlinear instructions. Thus, our solution, inspired by [114], is to encode disjunctions directly in the ACR without using disjunctive constraints, but rather taking advantage of the disjunctive nature of procedures in the ACR. For example, we replace the nonlinear arithmetic instruction $x := e_1 * e_2$ by a call $op_*(\langle e_1, e_2\rangle, \langle x\rangle)$, and define the auxiliary abstract procedure op_* by several rules that cover all possible input and simulate the corresponding disjunction. Importantly, each rule of op_* uses only a conjunction of linear constraints.

Example 4.3. Figure 4.3 depicts the ACR program obtained by applying the new abstract compilation to the RBR of method `clog`. This ACR is almost identical to the one in Figure 4.2, except that the instruction $y := y * b$ is now abstracted to a call $op_*(\langle y_0, b_0\rangle, \langle y_1\rangle)$ which is defined in Figure 4.3 (at the bottom). The rules of op_* were carefully designed to partition its input domain, such that, the postcondition of each case propagates accurate information about constancy, equality and progression (e.g., multiplication by a constant). In particular, we

$clog(\langle x_0, b_0\rangle, \langle\rangle) \leftarrow$
  $if_c(\langle x_0, b_0\rangle, \langle\rangle)$.
$if_c(\langle x_0, b_0\rangle, \langle\rangle) \leftarrow$
  $b_0 \leq 1$.
$if_c(\langle x_0, b_0\rangle, \langle\rangle) \leftarrow$
  $b_0 \geq 2$,
  $y_0 = 1$,
  $while_c(\langle x_0, b_0, y_0\rangle, \langle\rangle)$.

$while_c(\langle x_0, b_0, y_0\rangle, \langle\rangle) \leftarrow$
  $y_0 \geq x_0$.
$while_c(\langle x_0, b_0, y_0\rangle, \langle\rangle) \leftarrow$
  $y_0 + 1 \leq x_0$,
  acquire(1),
  $op_*(\langle y_0, b_0\rangle, \langle y_1\rangle,$ ★
  $while_c(\langle x_0, b_0, y_1\rangle, \langle\rangle)$.

$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $x = 0$, $z = 0$. (a)
$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $y = 0$, $z = 0$. (b)
$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $x = 1$, $z = y$. (c)
$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $y = 1$, $z = x$. (d)
$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $x = -1$, $z = -y$. (e)
$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $y = -1$, $z = -x$. (f)
$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $x \geq 2, y \geq 2$, $z \geq 2x, z \geq 2y$. (g)
$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $x \geq 2, y \leq -2$, $z \leq -2x, z \leq 2y$. (h)
$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $x \leq -2, y \geq 2$, $z \leq 2x, z \leq -2y$. (i)
$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $x \leq -2, y \leq -2$, $z \geq -2x, z \geq -2y$. (j)

Figure 4.3: The ACR program obtained by our modified abstract compilation for the `clog` program. The instruction $y := y * b$ is abstracted to $op_*(\langle y_0, b_0\rangle, \langle y_1\rangle)$. Below, we show the rules of the auxiliary procedure op_*.

distinguish the cases in which $x = 0$ (constancy), $x = \pm 1$ (equality) and those in which $|x| > 1$ and $|y| > 1$ (progress). Note that in case (g) the constraints $z \geq 2x$ and $z \geq 2y$ are crucial for finding a logarithmic UBF for our example. ■

After the abstract compilation, which abstracts each nonlinear operation to a call to an auxiliary procedure, the value analysis infers which of the cases of this procedure are enabled in the context of that call. This is done using an abstract interpretation to infer a precondition that describes how each ACR procedure is called. Technically, this is done using a fixpoint algorithm that computes the least fixed point (LFP) of a corresponding abstract collecting semantics. To keep the analysis efficient, we use the non-disjunctive abstract domain of polyhedra. Using this precondition, the value analysis can obtain a more precise

$$
\begin{array}{rll}
clog(x_0, b_0) = & if_c(x_0, b_0) & \{\} \\
if_c(x_0, b_0) = & 0 & \{b_0 \leq 1\} \\
if_c(x_0, b_0) = & while_c(x_0, b_0, y_0) & \{b_0 \geq 2, y_0 = 1\} \\
while_c(x_0, b_0, y_0) = & 0 & \{y_0 \geq x_0\} \\
while_c(x_0, b_0, y_0) = & 1 + while_c(x_0, b_0, y_1) & \{y_0 + 1 \leq x_0, \underbrace{b_0 \geq 2, y_0 \geq 1}_{\text{pre}-op_*}, \underbrace{y_1 \geq 2 * y_0}_{\text{post}-op_*}\}
\end{array}
$$

Figure 4.4: CRS inferred for the `clog` method, using our value analysis.

postcondition for the nonlinear operation, as the least upper bound (lub) of the cases that are enabled. Once these pre and postconditions are computed, the CRS is generated as described in Section 2.4, except that preconditions are incorporated in the CRS as well. Then, the corresponding CRS can be solved by PUBS to infer a UBF, just as done in Section 2.5.

Example 4.4. Consider the ACR program of Figure 4.3. For the ACR procedure $op_*(\langle b_0, y_0\rangle, \langle y_1\rangle)$, the LFP algorithm infers the precondition $\varphi \equiv \{b_0 \geq 2, y_0 \geq 1\}$. For each rule of the op_* procedure, the algorithm computes the conjunction of φ with the constraints in the rule. A rule is enabled if this conjunction is not equal to *false*. Only the rules (d) and (g) are enabled, and their conjunctions are $\varphi_d \equiv \{y_0 \geq 2, b_0 \geq 1, y_1 = y_0\}$ and $\varphi_g \equiv \{y_0 \geq 2, b_0 \geq 2, y_1 \geq 2y_0, y_1 \geq 2b_0\}$, respectively. The algorithm computes the postcondition of $op_*(\langle b_0, y_0\rangle, \langle y_1\rangle)$ as the lub (convex hull) of φ_d and φ_g, which is $\varphi' \equiv \varphi \wedge \{y_1 \geq 2 * y_0\}$. From the ACR program, we now generate the CRS depicted in Figure 4.4. Note that the pre and postcondition of the auxiliary ACR procedure op_* are added to the second equation of the CR $while_c$. Then, PUBS is able to solve it to the UBF $clog^+(x_0, b_0) = \log_2(\mathsf{nat}(x_0) + 1)$ which is asymptotically precise. ■

Note that the original value analysis of COSTA follows a bottom-up strategy similar to that of [30], but for our method to handle nonlinear instructions, we need a top-down approach [40] in order to infer preconditions as well. Finally, note that in our abstraction we ignore the possibility of arithmetic overflow. Like other works in this field, we assume that overflow is an erroneous behaviour that should be handled independently in a previous step.

2 Experimental Evaluation

We implemented our value analysis in COSTA, for this (1) we implemented the top-down LFP algorithm that is described in the corresponding paper; and (2) modified the abstract compilation of nonlinear RBR operations so as to insert the auxiliary procedures in the ACR program. We use the *Parma Polyhedra Library* [24] for representing the corresponding abstract domains.

For our experiments, we used typical benchmarks that use nonlinear and bitvector operations. These benchmarks come from the literature on program analysis and from the JAVA standard libraries.

Each benchmark was analysed for termination and cost with respect to the *number of instructions* cost model, using COSTA with and without our extension. Without our extension COSTA failed on all benchmarks as expected, and with the extension it succeeded to prove termination and infer UBFs for all benchmarks. We observed that our value analysis is slower than the original one of COSTA, this is mainly due to the use of a top-down LFP algorithm, instead of the bottom-up one [10].

We also applied two other termination analysers for JBC, JULIA [123] and APROVE [60], on the same set of benchmarks. JULIA could not prove any of them terminating. APROVE could not prove termination of benchmarks that use bitvector operations, and proved termination of the rest. However, it required significantly more time (an order of magnitude larger).

These experiments confirm that, by using a value analysis that 1) abstracts nonlinear operations to disjunctions of linear cases, 2) encodes these disjunctions in the ACR programs as extra rule, 3) uses a non-disjunctive domain like polyhedra, and 4) follows a top-down LFP algorithm; we can greatly improve the precision of the value analysis of programs that use nonlinear operations.

3 Related Work

The value analysis of COSTA is based on Abstract Interpretation [48, 46], a theory for semantic-based program analysis, which allows systematic derivation of sound program analysers. This theory has been used to develop industrial analysers, such as ASTRÉE [49] and Julia [123].

Abstract Compilation. The notion of abstract compilation was first proposed in [70] as a syntactic transformation of programs into abstract programs, that can be then analysed for inferring a property of interest. Boucher [33] extends this notion to include optimisations on the abstract program.

Numerical Abstract Domains. Apart from the domain of polyhedra [67], the LFP algorithm of our method can also use any weakly relational abstract domain [25] to represent linear inequalities. The benefit of using them is that, by restricting the shape of the linear constraints, the abstract operations for these domains are computationally cheaper than those of polyhedra, so the LFP algorithm scales better. However, the disadvantage of doing this is that the results may not be precise enough. For instance, the inequality $y_1 \geq 2 * y_0$ from Exam-

ple 4.4, which is crucial for inferring a precise AUBF, cannot be represented in some of these domains.

We could represent some arithmetic nonlinear instructions, like the product, using an abstract domain of conjunctions of equalities [110] and inequalities [28] between multivariate polynomials of bounded degree. Also, we could use the method of [63] to lift an abstract domain of linear inequalities to constraints between logarithmic, exponential, radical and max expressions. However, these domains are computationally expensive.

The abstract domain of linear constraints between the absolute values of variables [37] could be used to abstract some nonlinear operations. For instance, we could abstract $\mathtt{z} := \mathtt{x}\ \mathtt{mod}\ \mathtt{y}$ to $|z| < |y| \wedge |z| \leq |x|$.

Abstract Interpretation with Disjunctive Information A fundamental point in our technique is how to handle disjunctive information, a problem for which many solutions exist. On the side of abstract domains, one can use powerset domains [24] to represent finite disjunctions of elements. However, in [114] the authors argue that operations with these domains do not scale. Instead, they propose to handle disjunctive information via a program transformation, which they call an elaboration. The basic idea is to replace each node and its disjunctive invariant in the original CFG, by several nodes in the elaborated CFG, each with an invariant that corresponds to a disjunct of the original one. Trace partitioning [108] splits the set of all traces in the program semantics in several subsets, where each subset admits a non-disjunctive invariant.

4 Further Reading

In this Chapter we have briefly described how to handle nonlinear operations in the abstract compilation and the value analysis of COSTA, with the goal of inferring precise bounds for programs that use such operations. However, in the article the value analysis is presented towards the goal of proving termination by means of synthesising ranking functions.

In the article, we describe how to apply the method to handle not only the product $\mathtt{z} = \mathtt{x} * \mathtt{y}$, but also the nonlinear operations of integer quotient $\mathtt{z} = \mathtt{x}/\mathtt{y}$, integer modulo $\mathtt{z} = \mathtt{x}\%\mathtt{y}$, and bitvector operations such as bitwise and $\mathtt{z} = \mathtt{x}\&\mathtt{y}$, bitwise or $\mathtt{z} = \mathtt{x}|\mathtt{y}$, left shift $\mathtt{z} = \mathtt{x} << \mathtt{y}$, and right shift $\mathtt{z} = \mathtt{x} >> \mathtt{y}$. For inferring the pre and postcondition of each auxiliary procedure, the article also describes an interprocedural LFP algorithm that follows a top-down strategy [40].

5 | Amortised and Beyond

In this chapter we explore the limits of the classical approach to cost analysis which is used in COSTA, i.e., the approach that first abstracts the input program to a CRS and then solves this CRS into UBFs. It is known that this approach might infer UBFs that are asymptotically less precise than the actual cost. As yet, it was assumed that this imprecision is due to the way CRSs are solved into UBFs. We show that this assumption is partially true, and identify the reason due to which CRSs cannot precisely model the cost of some programs, independently from the precision of the underlying components. Then, to overcome these limitations, we develop a new approach to cost analysis, that is based on the use of satisfiability modulo theory (SMT for short) solvers and quantifier elimination (QE for short) procedures. Our approach is developed in a context in which, in addition to acquiring resources, programs can release resources as well. This gives rise to the notion of peak-cost. Our results have a strong relation to amortised cost analysis. This contribution has been published in the following article

> DIEGO ESTEBAN ALONSO-BLAS AND SAMIR GENAIM. On the Limits of the Classical Approach to Cost Analysis. In Antoine Miné and David Schmidt, editors of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 7460 of *Lecture Notes in Computer Science*, pages 405–421. Springer, September 2012

which is attached to this dissertation at Page 233.

Overview

Our main goal in this dissertation is to improve the precision of COSTA in several directions. In this Chapter we focus on well-known precision issues of the classical approach to cost analysis, on which COSTA is based.

A core assumption in COSTA, and Wegbreit's classical approach [133] in general, is that the input of an ACR procedure determines both its output and its cost. This is reflected in the fact that each ACR procedure is abstracted into the following two *separate* pieces information: (1) a CR that models how the cost depends on the input; and (2) a value postcondition that approximates the relation between the input and the output of the ACR procedure. In Section 5.1 we show that this assumption is the source of some well-known precision issues of COSTA. In particular, we provide examples for which there is an implicit

```
1   //@requires m >= 0 && size >= 0
2   void main(int size , int m) {
3       for( ; m > 0 ; m-- ) {
4           while( size > 0  && coin() )
5               size = size - 1 ;          ★
6       }
7   }
```

Figure 5.1: JAVA code for our guiding example.

codependency between the output of an ACR procedure and its cost, which is crucial to infer precise UBFs. COSTA abstracts this output-cost codependency away when it transforms the ACR into the CRS, and thus introduces spurious evaluations that exist in the CRS but not in the ACR. This, in turn, leads to inferring asymptotically imprecise UBFs.

To solve this imprecision, we turn the implicit output-cost codependency into an explicit one by allowing UBFs to use the output parameters as well. We refer to such bounds as **net-cost** UBFs, which describe the cost of complete finite executions. In Section 5.2 we develop a novel technique for inferring such UBFs that is based on the use of QE and SMT solving. In Section 5.3 we consider another notion of cost, which we call the peak-cost, that estimates the amount of resources that a program can hold simultaneously. This is useful for programs that can release resources and not only acquire them. Importantly, it also complements the net-cost bounds to handle non-terminating executions. We develop a novel technique for inferring peak-cost UBFs that is also based on the use of QE and SMT solving. In Section 5.4 we discuss an interesting relation between our approach and that of *automated amortised analysis* [79]. Briefly, we show that the potential functions in amortised analysis are just a restricted case of net-cost UBFs. In Section 5.5 we discuss an experimental evaluation of the techniques. In Section 5.6 we overview related work, and in Section 5.7 we describe some further details that can be found in the article.

1 The Output-Cost Codependency

Let us start our discussion with an example that illustrates the precision problem, of the classical approach to cost analysis, that we are dealing with in this chapter. The example is adapted from [129].

Example 5.1. Figure 5.1 includes a JAVA method in which the outer `for` loop performs `m` iterations, and in each iteration the inner `while` loop decrements

(a) $for(\langle s_0, m_0\rangle, \langle\rangle) \leftarrow$
$m_0 = 0,$
$s_0 \geq 0.$

(b) $for(\langle s_0, m_0\rangle, \langle\rangle) \leftarrow$
$m_0 \geq 1,$
$s_0 \geq 0,$
$\underline{while(\langle s_0\rangle, \langle s_2\rangle)},$
$m_2 = m_0 - 1,$
$for(\langle s_2, m_2\rangle, \langle\rangle).$

(c) $while(\langle s_0\rangle, \langle s_1\rangle) \leftarrow$
$s_0 \geq 0,$
$s_1 = s_0.$

(d) $while(\langle s_0\rangle, \langle s_1\rangle) \leftarrow$
$s_0 \geq 1,$
$\mathsf{acquire}(1),$
$s_2 = s_0 - 1,$
$while(\langle s_2\rangle, \langle s_1\rangle).$

Figure 5.2: ACR program for our guiding example.

(a)	$for(s_0, m_0) = 0$	$\{m{=}0, s_0 \geq 0\}$
(b)	$for(s_0, m_0) = \underline{while(s_0)} + for(s_2, m_2)$	$\{m_0 \geq 1, \underline{s_0 \geq s_2 \geq 0}, m_2 = m_0{-}1\}$
(c)	$while(s_0) = 0$	$\{s_0 \geq 0\}$
(d)	$while(s_0) = 1{+}while(s_2)$	$\{s_0 \geq 1, s_2 = s_0 - 1\}$

Figure 5.3: CRS for our guiding example.

variable `size` an arbitrary number of times – assuming that method `coin` non-deterministically returns `true` or `false`. We are interested in inferring a UBF on the number of visits to Line 5 (marked with ★). Note that the precise UBF is in $\Theta(\texttt{size})$.

Using COSTA, we generate the ACR program depicted in Figure 5.2 and the CRS depicted in Figure 5.3 (for simplicity we skip the RBR). We omit procedure *main* since it only calls procedure *for*. In the ACR program, the output variable s_1 of *while* represents the value of `size` upon exit from the `while` loop. Note that when generating Equation (b) of the CRS, from Rule (b) of the ACR, the output parameter s_2 of the call to *while* is removed and a corresponding postcondition $s_0 \geq s_2 \geq 0$ is added to the equation's constraints. Applying PUBS on this CRS results in the UBFs $while^+(s_0) = s_0$, which is asymptotically precise, and $for^+(s_0, m_0) = s_0 * m_0$, which is asymptotically imprecise. Note that, for simplicity, in this Chapter we write UBFs without using the `nat` operator, however, we will guarantee that the expressions are always non-negative. ■

In order to overcome the precision problem of the above example, we first need to identify which component of COSTA is responsible for this imprecision. There are four possibilities: (1) the abstract compilation of the RBR into the ACR; (2) the value analysis that infers postconditions at the level of the ACR; (3) the transformation of the ACR into the CRS; (4) the resolution of the CRS into the final UBF. Next we rule out three of these four possibilities:

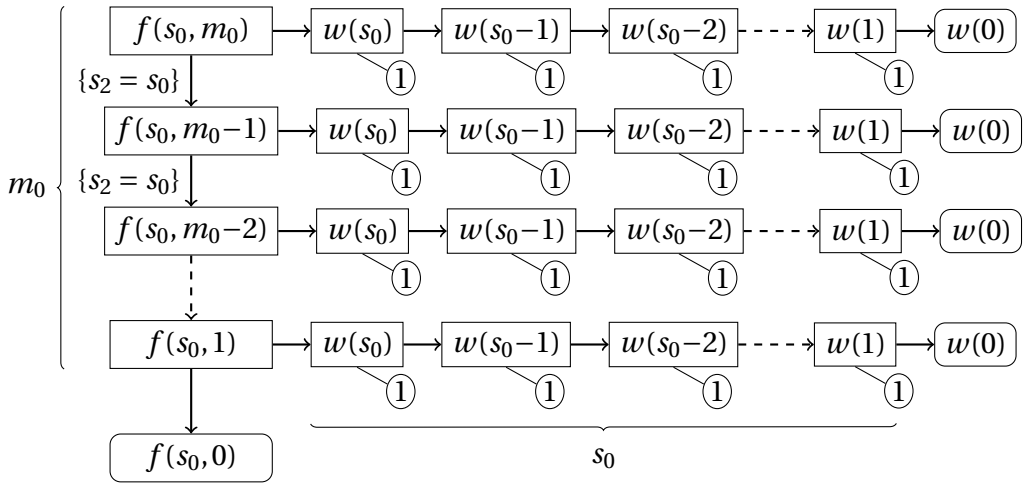


Figure 5.4: Evaluation tree of $for(s_0, m_0)$. We abbreviate *for*, *while* with f, w.

- The ACR in Figure 5.2 is a precise abstraction of the JAVA program of Figure 5.1. Namely, every ACR trace corresponds to an actual execution of `main`. Thus, the abstract compilation is ruled out;
- The post condition $s_0 \geq s_1 \geq 0$ for procedure *while*, which is inferred by the value analysis, is the most precise input-output relation for the `while` loop. Thus, the value analysis is ruled out as well; and
- Examining the CRS of Figure 5.3, one can see that any call to $for(s_0, m_0)$ has a corresponding evaluation tree as the one in Figure 5.4, which has a total cost of $s_0 * m_0$. This means that $for^+(s_0, m_0) = s_0 * m_0$ is a precise UBF for this CRS, so the resolution process of PUBS is ruled out as well.

The only component that has not been ruled out above is the one that transforms the ACR to the CRS, and thus it must be the one responsible for the above imprecision. Specifically, the CRS contains some spurious evaluations that correspond to no ACR trace, as we show in the next example.

Example 5.2. Consider again the CRS of Figure 5.3 and the evaluation tree of Figure 5.4, which correspond to an initial call $for(s_0, m_0)$. The evaluation tree includes m_0 nodes that correspond to applications of Equation (b) – the vertical chain on the left side. Each node has two out-edges that correspond to calling *while* and recursively calling *for*. Note that when recursively calling *for*, we choose a value s_2 for its first parameter such that $s_2 = s_0$ (this is explicitly written on the edges). It is important to note that this choice is valid according to the constraints attached to Equation (b). Each call to $while(s_0)$ creates a chain of recursive calls, which result in a sub-tree with s_0 nodes (the horizontal chains). These nodes correspond to applications of Equation (d), so each one

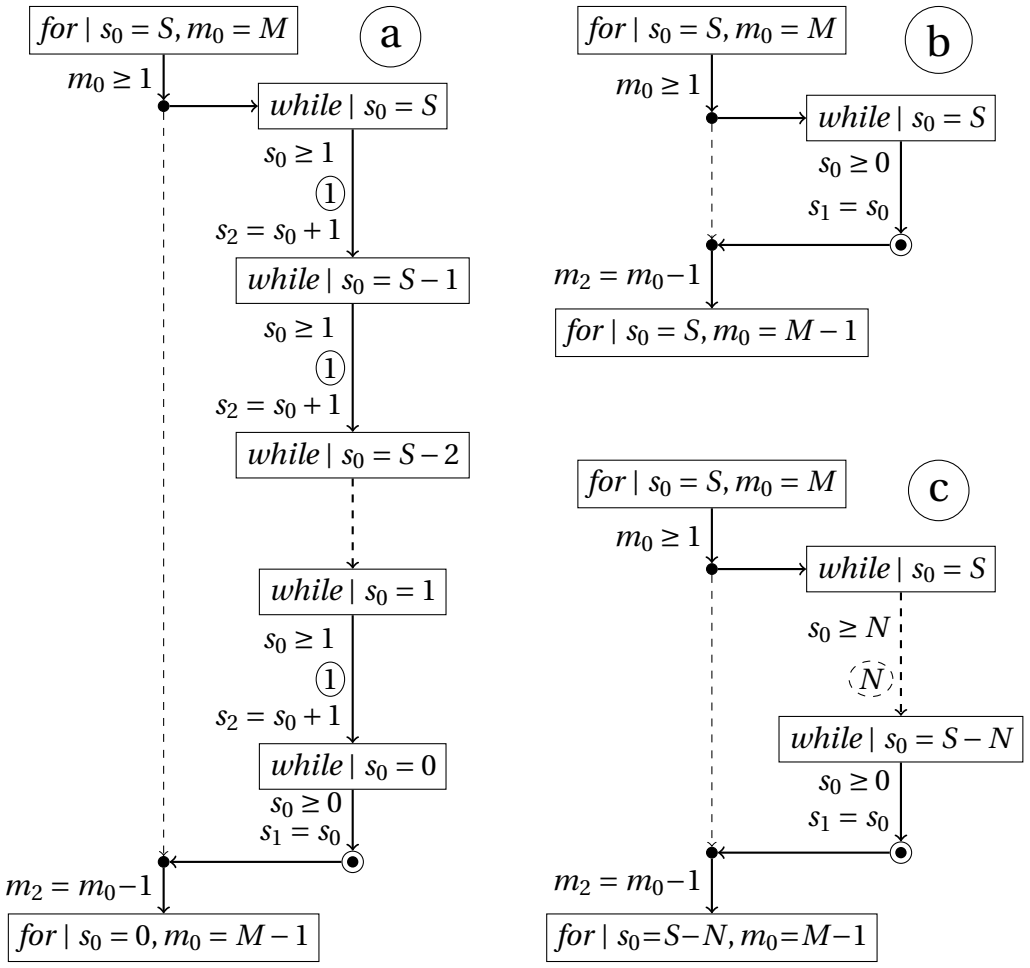


Figure 5.5: ACR trace fragments of a call to $for(\langle s_0\rangle, \langle m_0\rangle)$.

has a local cost 1 and thus the total cost of each such sub-tree is s_0. Since we have m_0 sub-trees, the total cost is $s_0 * m_0$.

Next we show why this evaluation tree is spurious, and thus the cost $s_0 * m_0$ as well, by comparing it to possible traces of the corresponding ACR. For this it is enough to consider only the first application of Equation (b), which corresponds to the top-most horizontal chain in Figure 5.4. Note that before applying the recursive call $for(s_2, m_0 - 1)$, the call $while(s_0)$ has already accumulated s_0 units to the cost. Note also that in the recursive call we have chosen a value for the first parameter s_2 such that $s_2 = s_0$. Let us see why this behaviour is not possible at the level of the ACR. Figure 5.5 depicts three ACR traces that correspond to the first iteration of $for(\langle s_0, m_0\rangle, \langle\rangle)$, these traces exhibit different behaviours depending on the behaviour of the call $while(\langle s_0\rangle, \langle s_2\rangle)$ which is

nondeterministic due to the use of method `coin`:

- Trace (a) corresponds to the case in which *while* makes s_0 iterations, so its cost is s_0 and upon exit $s_2 = 0$.
- Trace (b) corresponds to the case in which *while* makes 0 iterations, so its cost is 0 and upon exit $s_2 = s_0$.
- Trace (c) corresponds to a general case in which *while* makes N iterations, and thus the cost is N and upon exit $s_2 = s_0 - N$.

The evaluation tree of Figure 5.4 is spurious because, according to the above ACR traces, it is not possible that the cost of $while(\langle s_0 \rangle, \langle s_2 \rangle)$ is s_0 and at the same time to have $s_2 = s_0$ upon exit. Indeed, this behaviour is a combination of the traces (a) and (b). ■

The above example exposes a (somehow) surprising dependency between the cost of the ACR procedure *while* and its output, which turns out to be crucial to infer a precise UBF for procedure *for*. However, in COSTA, this dependency is lost when transforming the ACR Rule (b) into the CR Equation (b), mainly because output parameters are removed. In fact, the CR of *while* precisely models the relation between the cost and the input; and, the postcondition, which is used when removing the output parameters, precisely models the relation between the input and the output values, but none of them models the codependency between cost and output. This codependency can be modelled using a novel form of UBF that expresses the cost of a procedure in terms of both its input and output parameters.

Example 5.3. For procedure $while(\langle s_0 \rangle, \langle s_1 \rangle)$, the UBF $while^+(s_0) = s_0$, is the most precise UBF that depends only on its input parameter. However, the UBF $while^+(s_0, s_1) = s_0 - s_1$, in which we also use the output parameter s_1, describes the exact cost of *while*. ■

Defining a UBF in terms of the output parameters may seem counterintuitive. This is because a UBF is usually used to *statically* estimate the cost of the program, which typically corresponds to the amount of resources required for safely executing it. However, requiring information on the output parameters in order to evaluate the UBF is like actually requiring to execute the program. This is not really the case. When inferring UBFs, we distinguish between the entry procedure (e.g., *for*), and the rest (e.g., *while*). The UBF for the entry can always be defined in terms of its input parameters only, but to infer a precise UBF for the entry we may need to infer, for each other procedure, a UBF in terms of its input and output parameters. Moreover, later we will introduce the notion of peak-cost UBF which depends only on the input parameters, however net-cost UBFs are used when inferring it.

Inferring UBFs that depend on both input and output parameters requires a new form of CRSs that involve both kinds of parameters, instead of only the input ones. Due to the declarative nature of CRSs, a possible solution is to add the output parameters of each ACR procedure to the input parameters of its corresponding CR. However, PUBS would still fail to infer the desired UBF for our example, because its underlying methodology is based on multiplying the worst-case of all iterations of procedure *for* (which is s_0) by its number of iterations (which is m_0). Thus, in addition to enriching CRSs with output parameters, we also need a new resolution technique that is able to take into account the relation between costs of the different iterations of procedure *for*.

In the next sections we develop new techniques for computing UBFs, for several notions of cost, that are able to cope with the imprecision problems described above. Instead of using a new form of CRSs, we develop our techniques directly at the level of the ACR which, in principle, can be seen as a CRS with input and output parameters.

2 Inference of Net-Cost UBFs

Using the notation of Section 2.3, a complete finite execution trace t for a call $p(\bar{v},\bar{y})$ is one that starts in a state $\langle \bar{x}=\bar{v}, p(\bar{x},\bar{y})\rangle$ and ends in $\langle \psi,\epsilon\rangle$, where ϵ is an empty sequence of instructions. The net-cost of t is defined as the sum of the costs induced by its acquire(e_1) and release(e_2) instructions, it was denoted by $acrcost(t)$ in Section 2.3. The restriction to complete and finite execution traces is due to the use of output variables when inferring UBFs on the net-cost.

For programs that do not use release(e), the meaning of the net-cost coincide with the standard notion of cost, and can be used to estimate the amount of resources required to safely executing a program. However, if programs use release(e), then the usefulness of net-cost is not clear yet, since they clearly cannot be used to estimate the amount of resources required to safely executing a program. For example, the net-cost of a program that releases every resource that it acquires is 0. This will become clear in Section 5.3, for the rest of this section we may even assume that programs do not use release(e).

Our approach for inferring net-cost UBFs, that use both input and output parameters, is based on logical program analysis techniques. In essence, we view UBFs as program specifications, and then use the inductive assertion method [34, 81] to verify and synthesise those specifications. Our approach is thus developed in two steps:

1. **Verification**: In the first step, given an ACR program and a set of candidate net-cost UBFs, we develop a verification procedure to verify the validity

$$
\begin{array}{lll}
\widetilde{\phi}_a \equiv \forall s_0, m_0: & m_0{=}0 \wedge s_0 \geq 0 & \rightarrow \widetilde{f}(s_0, m_0) \geq 0 \\
\widetilde{\phi}_b \equiv \forall s_0, m_0, s_2, m_2: & \left(\begin{array}{l} m_0{\geq}1 \wedge s_0 \geq 0 \\ \wedge\, m_2 = m_0{-}1 \end{array}\right) & \rightarrow \widetilde{f}(s_0, m_0) \geq \widetilde{w}(s_0, s_2) + \widetilde{f}(s_2, m_2) \\
\widetilde{\phi}_c \equiv \forall s_0, s_1: & s_0 \geq 0 \wedge s_1 = s_0 & \rightarrow \widetilde{w}(s_0, s_1) \geq 0 \\
\widetilde{\phi}_d \equiv \forall s_0, s_1, s_2: & s_0 \geq 1 \wedge s_2 = s_0 - 1 & \rightarrow \widetilde{w}(s_0, s_1) \geq 1 + \widetilde{w}(s_2, s_1)
\end{array}
$$

Figure 5.6: Net Cost VC for the ACR program of Figure 5.2

of the given UBFs. This is done by deriving a **verification condition** (VC for short), which is a formula in First Order Logic (FOL for short), whose validity implies the validity of the candidate UBFs.

2. **Inference**: In a second step, we turn the verification procedure into an inference procedure that is able to synthesise net-cost UBFs, by using UBF **templates** and QE.

Next we explain these two steps by applying them to the example discussed in Section 5.1. The next example explains how to generate the VC.

Example 5.4. For the ACR program of Figure 5.2, we derive the VC $\widetilde{\Phi}$, which is a conjunction $\widetilde{\Phi} = \widetilde{\phi}_a \wedge \widetilde{\phi}_b \wedge \widetilde{\phi}_c \wedge \widetilde{\phi}_d$ of the clauses defined in Figure 5.6. The functions $\widetilde{w}(s_0, s_1)$ and $\widetilde{f}(s_0, m_0)$ refer to candidate net-cost UBFs for procedures *while* and *for*, respectively. As expected, $\widetilde{w}(s_0, s_1)$ uses both the input and the output parameters of the ACR procedure *while*.

Each clause $\widetilde{\phi}_i$ is generated from the ACR rule with label (i) in Figure 5.2, and has the form $\forall \bar{x} : \varphi \rightarrow f \geq g$. The inequality $f \geq g$ states that f, the net-cost of the corresponding procedure, is greater than (or equal to) the sum g of the net-cost induced by each instruction in the corresponding rule body. To derive the sum g, we let the net-cost of acquire(e) be e, that of release(e) be $-e$, that of a procedure call be its candidate net-cost UBF, and that of a constraint be 0. The clause states that $f \geq g$ must hold in a context φ, which is the conjunction of all constraints in the corresponding rule. The universal quantifier means that the inequality must hold for all valid valuations (of the program variables) that satisfy φ.

For instance, clause $\widetilde{\phi}_b$ states that for the function $\widetilde{f}(s_0, m_0)$ to be a valid UBF on the net-cost of procedure *for*, it has to be at least as the net-cost $\widetilde{w}(s_0, s_2)$ of the call to procedure *while*, plus the net-cost $\widetilde{f}(s_2, m_2)$ of the recursive call to procedure *for*. Moreover, this must hold for any values of s_0 and m_0 that satisfy the constraint $m_0 \geq 1 \wedge s_0 \geq 0 \wedge m_2 = m_0 - 1$ of Rule (b). ■

Given concrete definitions for the candidate net-cost UBFs, we can verify their validity simply by substituting them in the VC and then check its validity using, for example, an SMT solver.

Example 5.5. Let $\widetilde{f}(s_0, m_0){=}s_0$ and $\widetilde{w}(s_0, s_1){=}s_0{-}s_1$, which are the optimal UBFs on the net-cost of procedures *for* and *while*, respectively. Substituting these functions in the VC of Figure 5.6 results in:

$$\begin{array}{lll}
\widetilde{\phi}_a \equiv \forall s_0, m_0\colon & m_0{=}0 \wedge s_0 \geq 0 & \rightarrow s_0 \geq 0 \\
\widetilde{\phi}_b \equiv \forall s_0, m_0, s_2, m_2\colon & m_0{\geq}1 \wedge s_0 \geq 0 \wedge m_2 = m_0{-}1 & \rightarrow s_0 \geq s_0 - s_2 + s_2 \\
\widetilde{\phi}_c \equiv \forall s_0, s_1\colon & s_0 \geq 0 \wedge s_1 = s_0 & \rightarrow s_0 - s_1 \geq 0 \\
\widetilde{\phi}_d \equiv \forall s_0, s_1, s_2\colon & s_0 \geq 1 \wedge s_2 = s_0 - 1 & \rightarrow s_0 - s_1 \geq 1 + s_2 - s_1
\end{array}$$

Since the formula $\widetilde{\Phi} = \widetilde{\phi}_a \wedge \widetilde{\phi}_b \wedge \widetilde{\phi}_c \wedge \widetilde{\phi}_d$ is valid, then $\widetilde{f}(s_0, m_0)$ and $\widetilde{w}(s_0, s_1)$ are valid UBFs. On the other hand, if instead of $\widetilde{w}(s_0, s_1) = s_0 - s_1$ we use $\widetilde{w}(s_0, s_1) = s_0$, which is the most precise "input only" UBF for *while*, we get:

$$\begin{array}{lll}
\widetilde{\phi}_a \equiv \forall s_0, m_0\colon & m_0{=}0 \wedge s_0 \geq 0 & \rightarrow s_0 \geq 0 \\
\widetilde{\phi}_b \equiv \forall s_0, m_0, s_2, m_2\colon & m_0{\geq}1 \wedge s_0 \geq 0 \wedge m_2 = m_0{-}1 & \rightarrow s_0 \geq s_0 + s_2 \\
\widetilde{\phi}_c \equiv \forall s_0, s_1\colon & s_0 \geq 0 \wedge s_1 = s_0 & \rightarrow s_0 \geq 0 \\
\widetilde{\phi}_d \equiv \forall s_0, s_1, s_2\colon & s_0 \geq 1 \wedge s_2 = s_0 - 1 & \rightarrow s_0 \geq 1 + s_2
\end{array}$$

Now $\widetilde{\phi}_c$ and $\widetilde{\phi}_d$ are valid, since $\widetilde{w}$ is a valid UBF on the net-cost of *while*. However, $\widetilde{\phi}_b$ is not valid even though $\widetilde{f}(s_0, m_0) = s_0$ is a valid UBF on the net-cost of procedure *for*. This is because the validity of $\widetilde{f}$ cannot be proven using the "input only" UBF $\widetilde{w}(s_0, s_1) = s_0$. ■

Rather than verifying that some given candidates are valid, in automatic cost analysis the main interest is to directly synthesise those UBFs. This can be formulated as seeking net-cost UBFs $\{\widetilde{f}_1, \ldots, \widetilde{f}_k\}$ for which the corresponding VC is valid. However, this is a second order logical problem and solving it is impractical in general. A common approach to avoid solving a second order problem is to use template functions which restrict the form of functions that we are looking for [124, 127, 81, 128, 116]. A **template** is a function with a predefined structure, defined over the procedure parameters (as in the UBFs above), but contains as well some unknown template parameters. The use of such templates reduces the second order problem to that of seeking values for the unknown template parameters, which is a FOL problem.

Example 5.6. Consider the following UBF templates on the net-cost of procedures *for* and *while*:

$$\widetilde{f}(s_0, m_0) = f_{\mathrm{s}} s_0 + f_{\mathrm{m}} m_0 + f_{\mathrm{c}} \qquad \widetilde{w}(s_0, s_1) = w_{\mathrm{s}} s_0 + w_{\mathrm{t}} s_1 + w_{\mathrm{c}}$$

Note that s_0, m_0, s_1 are the parameters of the UBFs, which correspond to ACR variables, and $\{f_{\mathrm{s}}, f_{\mathrm{m}}, f_{\mathrm{c}}, w_{\mathrm{s}}, w_{\mathrm{t}}, w_{\mathrm{c}}\}$ are the template parameters. Substituting

these templates in the VC of Figure 5.6 results in the following FOL formulas:

$$\begin{array}{lll}
\widetilde{\phi}_a \equiv \forall s_0, m_0\colon & m_0{=}0 \wedge s_0 \geq 0 & \rightarrow f_\mathrm{s} s_0 + f_\mathrm{m} m_0 + f_\mathrm{c} \geq 0 \\
\widetilde{\phi}_b \equiv \forall \left\{\begin{smallmatrix} s_0, m_0, \\ s_2, m_2 \end{smallmatrix}\right\}\colon & \left(\begin{array}{l} m_0{\geq}1 \wedge s_0 \geq 0 \\ \wedge m_2 = m_0{-}1 \end{array}\right) & \rightarrow f_\mathrm{s} s_0 + f_\mathrm{m} m_0 \geq \left(\begin{array}{l} w_\mathrm{s} s_0 + w_\mathrm{t} s_2 + w_\mathrm{c} \\ + f_\mathrm{s} s_2 + f_\mathrm{m} m_2 \end{array}\right) \\
\widetilde{\phi}_c \equiv \forall s_0, s_1\colon & s_0 \geq 0 \wedge s_1 = s_0 & \rightarrow w_\mathrm{s} s_0 + w_\mathrm{t} s_1 + w_\mathrm{c} \geq 0 \\
\widetilde{\phi}_d \equiv \forall s_0, s_1, s_2\colon & s_0 \geq 1 \wedge s_2 = s_0 - 1 & \rightarrow w_\mathrm{s} s_0 + w_\mathrm{t} s_1 + w_\mathrm{c} \geq 1 + w_\mathrm{s} s_2 + w_\mathrm{t} s_1 + w_\mathrm{c}
\end{array}$$

In these formulas, the ACR variables are universally quantified but the template parameters are free.

Solving $\widetilde{\Phi}$ in this context means finding some values for $\{f_\mathrm{s}, f_\mathrm{m}, f_\mathrm{c}, w_\mathrm{s}, w_\mathrm{t}, w_\mathrm{c}\}$ for which the formula is valid. In turn, these values define instances of the templates that are valid UBFs. To solve $\widetilde{\Phi}$, we apply a QE procedure [81, §9], which eliminates the universally quantified variables (the ACR variables) and leaves an equivalent quantifier-free formula over the template parameters, namely $\widetilde{\Psi} = \widetilde{\psi}_a \wedge \widetilde{\psi}_b \wedge \widetilde{\psi}_c \wedge \widetilde{\psi}_d$, where:

$$\begin{array}{ll}
\widetilde{\psi}_a \equiv f_\mathrm{s} \geq 0 \wedge f_\mathrm{c} \geq 0 & \widetilde{\psi}_c \equiv\ w_\mathrm{s} + w_\mathrm{t} \geq 0 \wedge w_\mathrm{c} \geq 0 \\
\widetilde{\psi}_b \equiv f_\mathrm{s} \geq w_\mathrm{s} \wedge f_\mathrm{m} \geq w_\mathrm{c} \wedge f_\mathrm{s} + w_\mathrm{t} = 0 & \widetilde{\psi}_d \equiv w_\mathrm{s} \geq 1
\end{array}$$

The formulas $\widetilde{\Phi}$ and $\widetilde{\Psi}$ are equivalent, so any solution for $\widetilde{\Psi}$ is a solution for $\widetilde{\Phi}$. For instance, $\{f_\mathrm{s} = 1, f_\mathrm{m} = 0, f_\mathrm{c} = 0, w_\mathrm{s} = 1, w_\mathrm{t} = -1, w_\mathrm{c} = 0\}$ is a solution of $\widetilde{\Psi}$, which gives the net-cost UBFs $\widetilde{f}(s_0, m_0) = s_0$ and $\widetilde{w}(s_0, s_1) = s_0 - s_1$. ■

Inference of Lower Bounds. It is easy to adapt our approach to compute LBFs on the net-cost, simply by turning all $\geq$ to $\leq$ in the VC of Figure 5.6.

Net-cost and Non-Terminating Programs. Since the net-cost is defined for *finite* executions, the approach described above is not adequate for inferring the cost of non-terminating programs. This problem will be addressed and solved at the end of the next section.

3 Inference of Peak-Cost UBFs

The consumption of a resource that is only acquired, like the number of instructions, can be analysed using an **accumulative** cost model, that is, one that only introduces acquire(e) instructions in the ACR. However, some kinds of resources can be acquired and released during the execution, for instance, heap memory in the presence of a garbage collector (GC) [13]. Consumption of such resources is analysed using a **non-accumulative** cost model, which introduces in the ACR program instructions for both acquiring and releasing resources. To

```
1   //@requires n>=1
2   void p(int n) {
3     if (n > 1) {
4       int m = q(n);
5       // gc A
6       p(n-m);
7       // gc B
8     }
9   }
```

```
10  //@requires n>=2
11  int q(int n) {
12    int i = n/2;
13    do {
14      A x = new A();
15      B y = new B();
16      i--;
17    } while(i>0 && coin());
18    return n/2 - i;
19  }
```

Figure 5.7: JAVA code for the example on Peak heap space.

allow such cost models, we extend the ACR with an instruction release(e), which releases e resources, and the semantics with this rule:

$$\frac{eval(e,\psi) = v \geq 0}{\langle \psi, \mathsf{release}(e) \cdot \bar{a} \rangle \overset{-v}{\rightsquigarrow} \langle \psi, \bar{a} \rangle}$$

which annotates the transition with a negative value $-v$. Let us see an example of the usage of this ACR instructions.

Example 5.7. Consider the JAVA program depicted in Figure 5.7. Method `q` receives an integer `n`, executes a loop at least 1 and at most `n/2` iterations, and returns the number of iterations that it has performed. In each iteration it creates one object of class `A` and one of class `B`. Method `p` implements a loop (using recursion) where in each iteration it calls `q` with the current value of `n`, and then performs a recursive call with the loop counter decremented by `m` (the number of iterations that `q` has performed). We are interested in estimating the amount of heap space (in terms of the number of objects) required to run the program without running out of memory, assuming that the GC frees all instances of class `A` after the call to `q` (Line 5), and all instances of class `B` after the recursive call (Line 7). The corresponding ACR program is depicted in Figure 5.8. Note the use of release(m_2) to model the effect of applying the GC. ■

If a program is analysed with an accumulative cost model, then a net-cost UBF is enough to safely estimate how many resources are required to execute the program. However, if the program is analysed with a non-accumulative cost model, or if its semantics contains non-terminating executions, we then need to estimate its **peak-cost**, which is the maximum amount of resources held (i.e., acquired and not yet released) at any point of the execution. The peak cost of a

(a) $p(\langle n_0\rangle, \langle\rangle) \leftarrow$
$\quad n_0 = 1.$

(b) $p(\langle n_0\rangle, \langle\rangle) \leftarrow$
$\quad n_0 \geq 2,$
$\quad q(\langle n_0\rangle, \langle m_2\rangle),$
$\quad n_2 = n_0 - m_2,$
$\quad p(\langle n_2\rangle, \langle\rangle),$
$\quad \mathsf{release}(m_2).$

(c) $q(\langle n_0\rangle, \langle m_1\rangle) \leftarrow$
$\quad n_0 \geq 2,$
$\quad i_2 = n_0/2 - 1,$
$\quad \mathsf{acquire}(2),$
$\quad l(\langle i_2\rangle, \langle i_3\rangle),$
$\quad m_1 = n_0/2 - i_3,$
$\quad \mathsf{release}(m_1).$

(d) $l(\langle i_0\rangle, \langle i_1\rangle) \leftarrow$
$\quad i_0 \geq 0,$
$\quad i_1 = i_0.$

(e) $l(\langle i_0\rangle, \langle i_1\rangle) \leftarrow$
$\quad i_0 \geq 1,$
$\quad \mathsf{acquire}(2),$
$\quad i_2 = i_0 - 1,$
$\quad l(\langle i_2\rangle, \langle i_1\rangle).$

Figure 5.8: ACR program for the JAVA program of Figure 5.7

trace t is defined as $\max\{\, \mathit{acrcost}(t') \mid t' \text{ is a prefix of } t\}$. A function p^+ is a UBF for the peak-cost of procedure p, if for any input $\bar{v}$, for any trace t that starts in $\langle \bar{x} = \bar{v}, p(\bar{x}, \bar{y})\rangle$, and for any prefix t' of t, we have that $p^+(\bar{v}) \geq \mathit{acrcost}(t')$. Note that this definition covers both terminating and non-terminating executions.

Example 5.8. Consider the ACR program of Figure 5.8. Any execution of `p(n)` creates $2n$ objects, but all of them are released before the execution ends, so the net-cost of this program is 0 (this is what the analysis of Section 5.2 infers). However, due to the GC, the program can hold up to n objects at any moment. Thus, the function $\widehat{p}(n) = n$ is a UBF on the peak-cost of `p`. ■

In order to handle non-accumulative resources, we complement the UBF on the net-cost with a UBF on the peak-cost. A UBF on the peak-cost of an ACR procedure is a function over its input parameters such that, for any concrete input, any trace has a peak-cost below the UBF. Note that peak-cost UBFs are defined in terms of the input parameters only. This is because inferring them will require examining partial executions, and, moreover, they will also handle non-terminating executions, for which it is not possible to express the cost in terms of the output. However, we will use the net-cost UBFs, which are defined in terms of both input and output parameters, in order to infer peak-cost UBFs.

As in the case of the net-cost, our approach for inferring peak-cost UBFs is developed in two steps: given an ACR program and a candidate set of peak-cost UBFs, we first derive a VC whose validity implies the validity of the candidate UBFs; then we use UBF templates and QE to turn this verification procedure to an inference procedure.

Let us first explain the intuition behind the peak-cost VC. Given an ACR rule $p(\langle x\rangle, \langle y\rangle) \leftarrow q_1(\langle x\rangle, \langle w\rangle), q_2(\langle w\rangle, \langle y\rangle)$, when executing p, its peak-cost can be reached either *a*) during the execution of q_1; or *b*) during the execution of q_2. Case a) imposes the condition that the peak-cost of p is at least as the peak-cost of q_1, and Case b) imposes the condition that the peak-cost of p is at least as the

$$
\begin{array}{lll}
\widehat{\phi}_a \equiv\ \forall \bar{w}_a : n_0 = 1 & \rightarrow & \widehat{p}(n_0) \geq 0 \\
\widehat{\phi}_b \equiv\ \forall \bar{w}_b : (n_0 \geq 2 & \rightarrow & \widehat{p}(n_0) \geq 0 \quad) \\
\qquad \wedge\ (n_0 \geq 2 & \rightarrow & \widehat{p}(n_0) \geq \widehat{q}(n_0) \quad) \\
\qquad \wedge\ (n_0 \geq 2 \wedge n_2 = n_0 - m_2 & \rightarrow & \widehat{p}(n_0) \geq \widetilde{q}(n_0, m_2) + \widehat{p}(n_2) \quad) \\
\widehat{\phi}_c \equiv\ \forall \bar{w}_c : (n_0 \geq 2 \wedge n_0 = 2i_2 + 2 & \rightarrow & \widehat{q}(n_0) \geq 2 \quad) \\
\qquad \wedge\ (n_0 \geq 2 \wedge n_0 = 2i_2 + 2 & \rightarrow & \widehat{q}(n_0) \geq 2 + \widehat{l}(i_2) \quad) \\
\widehat{\phi}_d \equiv\ \forall \bar{w}_d : i_0 \geq 0 & \rightarrow & \widehat{l}(i_0) \geq 0 \\
\widehat{\phi}_e \equiv\ \forall \bar{w}_e : (i_0 \geq 1 & \rightarrow & \widehat{l}(i_0) \geq 2 \quad) \\
\qquad \wedge\ (i_0 \geq 1 \wedge i_2 = i_0 - 1 & \rightarrow & \widehat{l}(i_0) \geq 2 + \widehat{l}(i_2) \quad)
\end{array}
$$

$$
\begin{array}{lll}
\widetilde{\phi}_a \equiv\ \forall \bar{w}_a : n_0 = 1 & \rightarrow & \widetilde{p}(n_0) \geq 0 \\
\widetilde{\phi}_b \equiv\ \forall \bar{w}_b : n_0 \geq 2 \wedge n_2 = n_0 - m_2 & \rightarrow & \widetilde{p}(n_0) \geq \widetilde{q}(n_0, m_2) + \widetilde{p}(n_2) - m_2 \\
\widetilde{\phi}_c \equiv\ \forall \bar{w}_c : \left(\begin{array}{l} n_0 \geq 2 \wedge n_0 = 2i_2 + 2 \\ 2m_1 = n_0 - 2i_3 \end{array} \right) & \rightarrow & \widetilde{q}(n_0, m_1) \geq 2 + \widetilde{l}(i_2, i_3) - m_1 \\
\widetilde{\phi}_d \equiv\ \forall \bar{w}_d : i_0 \geq 0 \wedge i_1 = i_0 & \rightarrow & \widetilde{l}(i_0, i_1) \geq 0 \\
\widetilde{\phi}_e \equiv\ \forall \bar{w}_e : i_0 \geq 1 \wedge i_2 = i_0 - 1 & \rightarrow & \widetilde{l}(i_0, i_1) \geq 2 + \widetilde{l}(i_2, i_1)
\end{array}
$$

Figure 5.9: Net Cost and Peak Cost VC for the ACR program in Figure 5.8

amount of resources it holds upon exit from q_1, which coincides with the net-cost of q_1, plus the peak-cost of q_2. In addition, we require that the peak-cost is non-negative. Note the use of the net-cost in the second case. Let us now apply this intuition to our example.

Example 5.9. Using the ACR program of Figure 5.8 we derive the VC $\widehat{\Phi} = \widehat{\phi}_a \wedge \cdots \wedge \widehat{\phi}_e \wedge \widetilde{\phi}_a \wedge \cdots \wedge \widetilde{\phi}_e$, where $\widetilde{\phi}_i$ are clauses that correspond to the net-cost VC (derived as in Section 5.2) and $\widehat{\phi}_i$ are clauses for the peak-cost. As notation, $\bar{w}_i$ stands for the variables occurring at the formula indexed by i.

As explained above, the net-cost UBFs are required for formulating the peak-cost conditions. All clauses are depicted in Figure 5.9. Clause $\widehat{\phi}_i$ is generated from the ACR rule with label (i) in Figure 5.8, following the intuition described above. For example, clause $\widehat{\phi}_b$ includes three conjuncts: the first one states that the peak-cost of p is positive, the second one states that it is at least as that of q; and the third one states that it is at least the peak-cost of the recursive call to p plus the amount of resources held before that call. This last amount is exactly the net-cost $\widetilde{q}(n_0, m_2)$ of q. ■

Verifying the validity of candidate peak-cost UBFs is simply done by substituting them in the VC and checking for its validity. However, our interest is in inferring these UBFs rather than checking the validity of the given ones. This can be done using UBF templates and QE as in the case of net-cost.

$$
\begin{array}{ll}
\widehat{\psi}_a \equiv & \widehat{p}_n + \widehat{p}_c \geq 0 \\
\widehat{\psi}_b \equiv & (\widehat{p}_n \geq 0 \wedge 2\widehat{p}_n + \widehat{p}_c \geq 0) \ \wedge \ (\widehat{p}_n \geq \widehat{q}_n \wedge 2\widehat{p}_n + \widehat{p}_c \geq 2\widehat{q}_n + \widehat{q}_c) \\
& \quad \wedge \ (\widetilde{q}_n \leq 0 \wedge 2\widetilde{q}_n + \widetilde{q}_c \leq 0 \wedge \widehat{p}_n = \widetilde{q}_m) \\
\widehat{\psi}_c \equiv & (\widehat{q}_n \geq 0 \wedge 2\widehat{q}_n + \widehat{q}_c \geq 2) \ \wedge \ (2\widehat{q}_n \geq \widehat{l}_i \wedge 2\widehat{q}_n + \widehat{q}_c \geq \widehat{l}_c + 2) \\
\widehat{\psi}_d \equiv & \widehat{l}_i \geq 0 \wedge \widehat{l}_c \geq 0 \\
\widehat{\psi}_e \equiv & (\widehat{l}_i \geq 0 \wedge \widehat{l}_i + \widehat{l}_c \geq 2) \ \wedge \ (\widehat{l}_i \geq 2)
\end{array}
$$

$$
\begin{array}{ll}
\widetilde{\psi}_a \equiv & \widetilde{p}_c + \widetilde{p}_n \geq 0 \\
\widetilde{\psi}_b \equiv & \widetilde{p}_n = \widetilde{q}_m - 1 \wedge \widetilde{q}_n \leq 0 \wedge 2\widetilde{q}_n + \widetilde{q}_c \leq 0 \\
\widetilde{\psi}_c \equiv & 2\widetilde{q}_n \geq \widetilde{l}_i + \widetilde{l}_j \wedge \widetilde{q}_m + \widetilde{l}_j = -1 \wedge 2\widetilde{q}_n + \widetilde{q}_c \geq \widetilde{l}_j + \widetilde{l}_c + 2 \\
\widetilde{\psi}_d \equiv & \widetilde{l}_i + \widetilde{l}_j \geq 0 \wedge \widetilde{l}_c \geq 0 \\
\widetilde{\psi}_e \equiv & \widetilde{l}_i \geq 2
\end{array}
$$

Figure 5.10: Quantifier-Free clauses, obtained by substituting the template UBFs of Example 5.10 in the VC of Figure 5.9, and then applying real QE.

Example 5.10. Consider the following net- and peak-cost UBF templates for the ACR program of Figure 5.8:

$$
\begin{array}{lll}
\widehat{p}(n_0) = \widehat{p}_n n_0 + \widehat{p}_c & \widehat{q}(n_0) = \widehat{q}_n n_0 + \widehat{q}_c & \widetilde{q}(n_0, m_1) = \widetilde{q}_n n_0 + \widetilde{q}_m m_1 + \widetilde{q}_c \\
\widetilde{p}(n_0) = \widetilde{p}_n n_0 + \widetilde{p}_c & \widehat{l}(i_0) = \widehat{l}_i i_0 + \widehat{l}_c & \widetilde{l}(i_0, i_1) = \widetilde{l}_i i_0 + \widetilde{l}_j i_1 + \widetilde{l}_c
\end{array}
$$

Substituting these templates in the VC of Figure 5.9, and then eliminating the universal quantifiers results in the quantifier free formulas $\widehat{\Psi} = \widehat{\psi}_a \wedge \cdots \wedge \widehat{\psi}_e$ and $\widetilde{\Psi} = \widetilde{\psi}_a \wedge \cdots \wedge \widetilde{\psi}_e$ where $\widetilde{\psi}_i$ and $\widehat{\psi}_i$ are depicted in Figure 5.10. Each clause $\widehat{\psi}_i$ (resp. $\widetilde{\psi}_i$) is obtained by eliminating the universal quantifier from the clause $\widehat{\phi}_i$ (resp. $\widetilde{\phi}_i$). Every solution for $\widehat{\Psi} \wedge \widetilde{\Psi}$ defines instances of the templates that are valid UBFs. In particular, the following solution:

$$
\begin{array}{lllllllll}
\widehat{p}_n = 1 & \widehat{p}_c = 0 & & \widehat{q}_n = 1 & \widehat{q}_c = 0 & & \widetilde{q}_n = 0 & \widetilde{q}_m = 1 & \widetilde{q}_c = 0 \\
\widetilde{p}_n = 0 & \widetilde{p}_c = 0 & & \widehat{l}_i = 2 & \widehat{l}_c = 0 & & \widetilde{l}_i = 2 & \widetilde{l}_j = -2 & \widetilde{l}_c = 0
\end{array}
$$

defines the following UBFs on the peak and net-cost:

$$
\begin{array}{lll}
\widehat{p}(n_0) = n_0 & \widehat{q}(n_0) = n_0 & \widetilde{q}(n_0, m_1) = m_1 \\
\widetilde{p}(n_0) = 0 & \widehat{l}(i_0) = 2i_0 & \widetilde{l}(i_0, i_1) = 2i_0 - 2i_1
\end{array}
$$

Note that for procedure q we use a net-cost UBF that only depends on the output parameter m_1 of q. ■

Let us finish this section by commenting on the case of non-terminating programs.

Example 5.11. Consider a non-terminating ACR program defined by the rule

$$p(\langle n_0\rangle, \langle\rangle) \leftarrow n_0 \geq m \geq 0, \mathsf{acquire}(m), n_2 = n_0 - m, p(\langle n_2\rangle, \langle\rangle).$$

Procedure p receives a non-negative integer n_0, nondeterministically chooses a non-negative value $m \leq n_0$, acquires m resources, and then calls p recursively with $n_0 - m$. The peak-cost of this program is exactly n_0, since no trace can acquire more than n_0 resources, and there are some traces that acquire exactly n_0. This peak-cost UBF can be inferred using the approach described in this section. For more details, see Example 9 of the article. ■

4 Relation to Amortised Cost Analysis

Let us discuss now an interesting relation we observed between UBFs that are defined in terms of both input and output parameters, and the notion of potential functions used in amortised cost analysis. This offers a semantics-based explanation to why amortised analysis can obtain more precise UBFs.

In the context of an ACR program, a potential function maps a given state to a non-negative number, which is called the potential of the state. This potential can be interpreted as the amount of resources available in the given state. An automatic amortised cost analysis [79, 73] assigns to each ACR procedure $p(\bar{x}, \bar{y})$ two potential functions: one for the input $P_p(\bar{x})$, and one for the output $Q_p(\bar{y})$. Intuitively, the input potential $P_p(\bar{x})$ must be large enough to pay for the cost of executing $p(\bar{x}, \bar{y})$, and, upon exit, leave at least $Q_p(\bar{y})$ units to pay the cost of the rest of the execution. Thus, if c is the net-cost of p, then $P_p(\bar{x}) \geq c + Q_p(\bar{y})$ must hold. This can be rewritten as $P_p(\bar{x}) - Q_p(\bar{y}) \geq c$, so the difference $P_p(\bar{x}) - Q_p(\bar{y})$ is a UBF on the net-cost of p, defined both over the input and output of p.

The above potential functions are in principle UBFs as described in Section 5.2, however, they are just a special case. We have observed some cases in which the net-cost UBF cannot be expressed as the difference of two functions as above, i.e., one over the input parameters and one over the output parameters, but rather it includes a monomial that combines input and output parameters. For more related details, see Section 6 in the article. It is worth noting, in addition, that net-cost UBFs are not limited to polynomial templates as the case of the potential functions used in [73].

5 Experimental Evaluation

Implementation. We have developed for our method a prototype implementation, called ACRP. This program takes less than 1500 lines of HASKELL code.

ACRP takes as input a text file that, in a PROLOG-like syntax, contains an ACR program and a template UBF on the net and peak cost of each procedure. ACRP generates the VC for the net and peak cost of the ACR program. Then, ACRP invokes the QE methods provided by REDLOG [56] (an extension of the REDUCE computer algebra system [68]) to generate the constraints over the parameters of the template UBFs. ACRP uses the FOL theory of *real closed fields*, which allows us to use a wide range of template UBFs, such as multivariate polynomials and max and min operations. ACRP includes also an option to use the combination of REDLOG, QEPCAD and SLFQ of [127], to reduce the size of the constraints. ACRP outputs the constraint as a SMTLIB2 script [29], using the logic of *nonlinear real arithmetic* (`QF_NRA`). This script can then be passed to an SMT solver like Z3 [53], to obtain a solution of the parameters.

Precision Experiments. We have applied the analyser on small ACR programs that we wrote from examples in the literature. Some of these examples are the methods to manipulate a binary counter [45, §17], or some nested loops analysed in the articles of the SPEED project [65, 66, 64], For these examples we obtained the expected precise UBFs. In many of them, there was one procedure for which it was necessary to use a UBF on the net cost that depended on the output parameters. This indicates the presence of the output-cost codependency that we mentioned in many of these programs, and the effectiveness of our approach to cost analysis. However, we also found that some programs usually described in terms of amortised cost, like the methods for a dynamic array [45, §17] or for a queue implemented with two stacks [103], could be solved *without* using UBFs that depended on the output parameters. This indicated us that the output-cost codependency is not important for such programs, and that these could in principle be solved with precision using a more precise CR solver. This was an inspiration of our contribution described in Chapter 6.

Scalability Limitations. Unfortunately, due to the high computational costs of QE procedures for the theory of real closed fields, ACRP can only be applied to small examples and it does not scale for large programs. This limits the practicality of our approach, however, it does not degrade its theoretical importance. Besides, our approach is applicable to any set of template UBFs, so long as they are supported by the QE procedure. In this sense, the choice of a QE procedure and a FOL theory determines the scope and the scalability of the technique. In particular, in Chapter 6 we overcome these limitations by restricting ourselves to scalable QE procedures for linear real arithmetic.

6 Related Work

Automated Amortised Analysis. In Section 5.4, we have already commented on the connection between our approach and that of automated amortised analysis. This approach was introduced in the doctoral dissertation of Jost [79], and has later been extended in several directions [35, 80, 122, 76, 73].

Inductive Assertion Method. Our approach for inferring UBFs on the net- and peak-cost is based on viewing these UBFs as program specifications, and then use the inductive assertion method to verify and synthesise those specifications. This idea was already proposed by Wegbreit [134] for verifying average-cost estimates. QE over the real field has been previously used for inferring UBFs [19], and for verifying UBFs [52].

Simple Loops. The works of the SPEED project [64, 66, 140, 65], present analysis techniques for imperative programs that are able to handle a particular form of the output-cost codependency, which appears in some nested loops in which the control variable of an outer loop is modified inside an inner loop. Unlike ours, their approach cannot handle multiple recursion.

7 Further Reading

In this Chapter we have described the technical contributions of [18], which studies the limitations of the classical approach to cost analysis on which COSTA is based. We explained why the source of these limitations is related to the fact that CRSs ignore the output parameters of ACR procedures, and showed that output-cost codependency is crucial for inferring precise UBFs. In order to overcome these limitations, we introduced the notion of net-cost UBFs, which are UBFs defined in terms of both input and output parameters. We have also developed a novel technique for inferring such UBFs, and extended them to infer UBFs on the peak-cost of programs. We have observed a strong relation to amortised cost analysis [73], which provides an alternative semantic-based explanation to why amortised analysis (of ACR programs) can be more precise than the classical approach.

The article extends and formalises the contents of this chapter. It includes a formal syntax and semantics for the ACR language, as well as the definitions of net-cost and peak-cost. It also describes the derivation of the different VCs from the ACR program, and provides correctness statements together with their corresponding proofs. The article also contains the tables with the results of our experiments.

6 | Logical Resolution of CRS

In this chapter we develop a precise and scalable technique for solving CRSs into UBFs. This is done by exploring the gap between the scalable technique of PUBS, and the precise techniques of Chapter 5. Our technique first splits the input CRS into several atomic ones, uses precise local reasoning to infer the corresponding UBFs, and then combines these UBFs into a UBF for the original CRS. For the local reasoning we propose several methods that define the cost as a solution of a universally quantified FOL formula, similarly to what has been done in Chapter 5. We also rely on techniques used in PUBS. This contribution has been published in the following article:

> DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Precise Cost Analysis via Local Reasoning. In Dang Van Hung and Mizuhito Ogawa, editors of the Proceedings of *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 8172 of *Lecture Notes in Computer Science*, pages 319–333. Springer, October 2013

which is attached to this dissertation at Page 253.

Overview

The techniques used in PUBS [5] to compute UBFs are based on assuming worst-case behaviour for all equations, both for the cost contributed by each equation and for the number of times each equation is applied. This approach is efficient and can handle a wide class of CRSs. However, it might lead to undesired imprecision, which might be even unrelated to output-cost codependency of Chapter 5. This imprecision becomes even more significant for CRSs that originate from divide and conquer algorithms. In such CRSs, the number of times each equation is applied and the cost contributed by each application are not independent, so taking the worst-case of each measure leads to imprecise AUBFs.

The imprecision of PUBS, among other issues, was addressed in Chapter 5 where precise and novel techniques for solving CRSs were proposed. They are based on defining the cost as a solution of a corresponding universally quantified FOL formula. This method, as expected, would obtain the most precise UBFs, however, it has two major limitations: (1) template UBFs have to be

provided by the user; and (2) the use of QE renders the technique impractical, since QE is computationally expensive in the general case.

In this chapter we explore the gap between these two approaches, looking for solving techniques with an efficiency close to those of PUBS and a precision close to those of Chapter 5. Concretely, we develop a novel technique that splits the input CRS into atomic CRs of a simpler form, solves each of them separately, and then combines the resulting UBFs into UBFs for the original CRS. We call a CR atomic, if all equations contribute 0, except one equation that contributes a basic cost expression, i.e, of the form m, $\mathsf{nat}(l)$, $\log_m(\mathsf{nat}(n)+1)$, or $m^{\mathsf{nat}(n)}-1$. Our main observation is that it is enough to solve few atomic CRs precisely, using techniques similar to those of Chapter 5, while solving the others as in PUBS without affecting the overall precision.

In Sections 6.1 and 6.2 we propose two methods for precisely solving atomic CRs, which are based on the idea of specifying the cost using universally quantified FOL formulas as in Chapter 5. However, we do not require the user to provide any template (we always use linear templates), and, importantly, the generated VCs have almost a linear form for which QE can be done efficiently. In Section 6.3, we show how to handle the general case by reducing the problem of solving a given CRS into that of solving several atomic CRs. In Section 6.4 we discuss an experimental evaluation of the techniques. In Section 6.5 we overview related work, and in Section 6.6 we describe some further details that can be found in the article.

1 The Tree-Sum Method

In this section we describe the tree-sum method for solving atomic CRs. We first assume that the non-zero basic cost expression in the atomic CR is of the form $\mathsf{nat}(l)$, then, at the end of this section, we describe how to handle arbitrary basic cost expressions. We start with a motivating example for which the underlying techniques [5, 14] of PUBS infer an asymptotically imprecise UBF.

Example 6.1. Consider a CR C defined by the following equations:

$$\begin{array}{ll} C(m) = 0 & \{m = 0\} \\ C(m) = \mathsf{nat}(n) + C(m-n) & \{m \geq n, n \geq 1\} \end{array}$$

Note that each application of the recursive equation contributes n (between 1 and m) units to the cost, and that the same amount n is subtracted from the input m in the recursive call. Figure 6.1 depicts three evaluation trees for $C(m)$:

- In tree (a), the chain of recursive calls is the largest possible. In each node, the contributed cost is $n = 1$, which allows applying the recursive equation m times. Thus, the total cost is m;

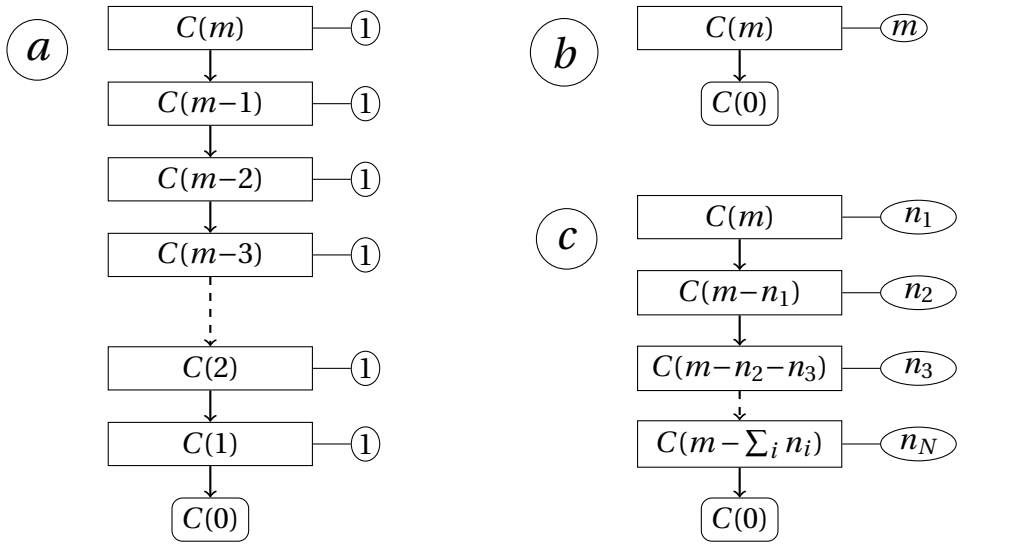


Figure 6.1: Evaluation trees for $C(m)$, with (a) maximum length, (b) maximum local cost, and (c) anything between them.

- In tree (b), the local cost of the first call is chosen as $n = m$, which prevents further applications of the recursive equation. The total cost is also m; and
- In tree (c), the recursive equation is applied N times, where $1 \leq N \leq m$. The i-th node, which corresponds to the i-th application, contributes n_i units to the cost such that $m = \sum_{i=1}^{N} n_i$. Still the total cost is also m.

From these trees, in particular tree (c), we can conclude that every evaluation of $C(m)$ has a total cost m. Thus, the most precise UBF is $C^+(m) = \mathsf{nat}(m)$. However, PUBS computes the UBF $C^+(m) = \mathsf{nat}(m)^2$ which is asymptotically less precise. This is because it multiplies the worst-case cost of all nodes, which is m as in Tree (b), by the maximum possible number of internal nodes, which is also m as in Tree (a). ■

In order to precisely handle examples like the one above, we should take into account the relation between the cost contributed in each node of an evaluation tree and the number of nodes in the same tree. This way, we can rule out (some) cases in which the worst-case node cost and worst-case number of nodes come from different trees. None of the techniques on which PUBS is based [5, 14] is able to take this relation into account. In what follows we describe an approach that is able to model this relation.

Our approach is based on the inductive assertion method for proving program correctness [34], like the the approach that we have described in Section 5.2. In particular, it is based on deriving a VC, from the CR this time, whose

validity implies the validity of some candidate UBFs. However, despite this similarity, in the case of atomic CRs solving the VC can be efficient as well.

Example 6.2. Given the CR of Example 6.1, and a candidate UBF $C^+(m)$, the corresponding VC is a conjunction of the following clauses:

$$\begin{array}{lll} \forall m: & m = 0 & \rightarrow \quad C^+(m) \geq 0 \\ \forall m,n: & m \geq n \geq 1 & \rightarrow \quad C^+(m) \geq \mathsf{nat}(n) + C^+(m-n) \end{array}$$

The first clause corresponds to the first equation of the CR, and the second one to the second equation. Intuitively, each clause requires that the candidate UBF C^+ covers the local cost of the corresponding equation, and the cost of the recursive calls. The similarity to the net-cost formulas of Section 5.2 is clear. ■

In order to check that a given cost expression $f(m)$ is a valid UBF, we simply replace $C^+(m)$ in the VC of Example 6.2 by $f(m)$, and then check for its validity. However, as in Chapter 5, our interest is in synthesizing a UBF rather than checking the validity of a given one. This can be done as in Chapter 5 by using UBF templates, together with QE and SMT solving.

Example 6.3. Let $f(m) = \mathsf{nat}(a_{\mathrm{m}} m + a_{\mathrm{c}})$ be a UBF template for the CR of Example 6.1. Note that a_{m} and a_{c} are the template parameters. Substituting this template in the VC of Example 6.2 results in a FOL formula that is the conjunction of the following clauses:

$$\begin{array}{lll} \forall m: & m = 0 & \rightarrow \mathsf{nat}(a_{\mathrm{m}} m + a_{\mathrm{c}}) \geq 0 \\ \forall m,n: & m \geq n \geq 1 & \rightarrow \mathsf{nat}(a_{\mathrm{m}} m + a_{\mathrm{c}}) \geq \mathsf{nat}(n) + \mathsf{nat}(a_{\mathrm{m}}(m-n) + a_{\mathrm{c}}) \end{array}$$

Note that the parameters $a_{\mathrm{m}}, a_{\mathrm{c}}$ are free variables. Any satisfying assignment for this formula defines a template instance that is a UBF, e.g., $a_{\mathrm{m}} = 1, a_{\mathrm{c}} = 0$ and $a_{\mathrm{m}} = 2, a_{\mathrm{c}} = 1$ define $f(m) = \mathsf{nat}(m)$ and $f(m) = \mathsf{nat}(2m+1)$, respectively. As in Section 5.2, solving the VC means finding such an instance automatically. Again, we first apply a QE procedure to eliminate the variables n and m, which results in the equivalent quantifier-free constraint $a_{\mathrm{m}} \geq 1 \wedge a_{\mathrm{c}} \geq 0$. Clearly, the instances we mentioned above are solutions of this constraint. Next, we apply an SMT solver to obtain a particular solution of this constraint. ■

Note that the SMT may give a solution like $a_{\mathrm{m}} = 2 \wedge a_{\mathrm{c}} = 5$ that corresponds to a non-optimal UBF. In order to find an optimal assignment, one may use heuristics based on linear programing, or a greedy strategy like [65].

As we have discussed in Chapter 5, QE procedures can be computationally expensive in general. However, if we restrict ourselves to linear UBF templates, i.e., of the form $\mathsf{nat}(l)$ where l is a linear expression with parametric coefficients, as in Example 6.3, then the derived VC is of a particular form that is

almost linear. This is because we have restricted ourselves to atomic CRs with cost expressions of the form $\mathsf{nat}(l')$ as well. Fortunately, VCs of this form, unlike the general form used in Chapter 5, can be solved (or precisely approximated) using linear programming techniques that have polynomial-time complexity. This renders the overall approach practical. The article contains more details on how to solve such VCs.

Handling Arbitrary Basic Cost Expressions. Now we extend the above technique to handle basic cost expression b, i.e., $b = m$, $b = m^{\mathsf{nat}(l)} - 1$, or $b = \log_m(\mathsf{nat}(l) + 1)$, instead of only $b = \mathsf{nat}(l)$. Note, however, that we still require that only one equation has a non-zero cost expression. The case of $b = m$ is simply handled by considering it as $\mathsf{nat}(m)$, thus we concentrate on the other two cases. A possible solution is to simply generate the VC with these basic cost expressions, however, this will result in a VC that does not have the desired (almost linear) form, and thus solving it might be computationally expensive. Instead, our approach for solving such CRs is as follows: we first replace the basic cost expression ($m^{\mathsf{nat}(l)} - 1$ or $\log_m(\mathsf{nat}(l) + 1)$) by its sub-expression $\mathsf{nat}(l)$. This results in a new CR E that can be solved as described above to a UBF $E^+(\bar{x}) = \mathsf{nat}(L)$. Using this UBF, we then generate a UBF for C as follows:

$$C^+(\bar{x}) = \begin{cases} 1.5 * \mathsf{nat}(L) & \text{if } b = \log_m(\mathsf{nat}(l) + 1) \\ m^{\mathsf{nat}(L)} - 1 & \text{if } b = m^{\mathsf{nat}(l)} - 1 \end{cases}$$

Example 6.4. Consider the CR C defined by the following equations:

$$\begin{array}{ll} C(m) = 0 & \{m = 0\} \\ C(m) = 2^{\mathsf{nat}(n)} - 1 + C(m-n) & \{m \geq n, n \geq 1\} \end{array}$$

We replace $2^{\mathsf{nat}(n)} - 1$ by $\mathsf{nat}(n)$, resulting in the follwoing CR:

$$\begin{array}{ll} E(m) = 0 & \{m = 0\} \\ E(m) = \mathsf{nat}(n) + E(m-n) & \{m \geq n, n \geq 1\} \end{array}$$

We can solve this CR as described above to the UBF $E^+(m) = \mathsf{nat}(m)$. Then we define $C^+(m) = 2^{\mathsf{nat}(m)} - 1$ as a UBF for C. Note that applying PUBS on this CR we obtain the AUBF $C^+(m) = \mathsf{nat}(m) * 2^{\mathsf{nat}(m)}$, which is imprecise. ■

2 The Level-Sum Method

The technique of the previous section handles cases for which the given atomic CR admits a linear UBF. In this section, building on similar ideas, we develop an approach for solving atomic CRs that exhibit a divide and conquer

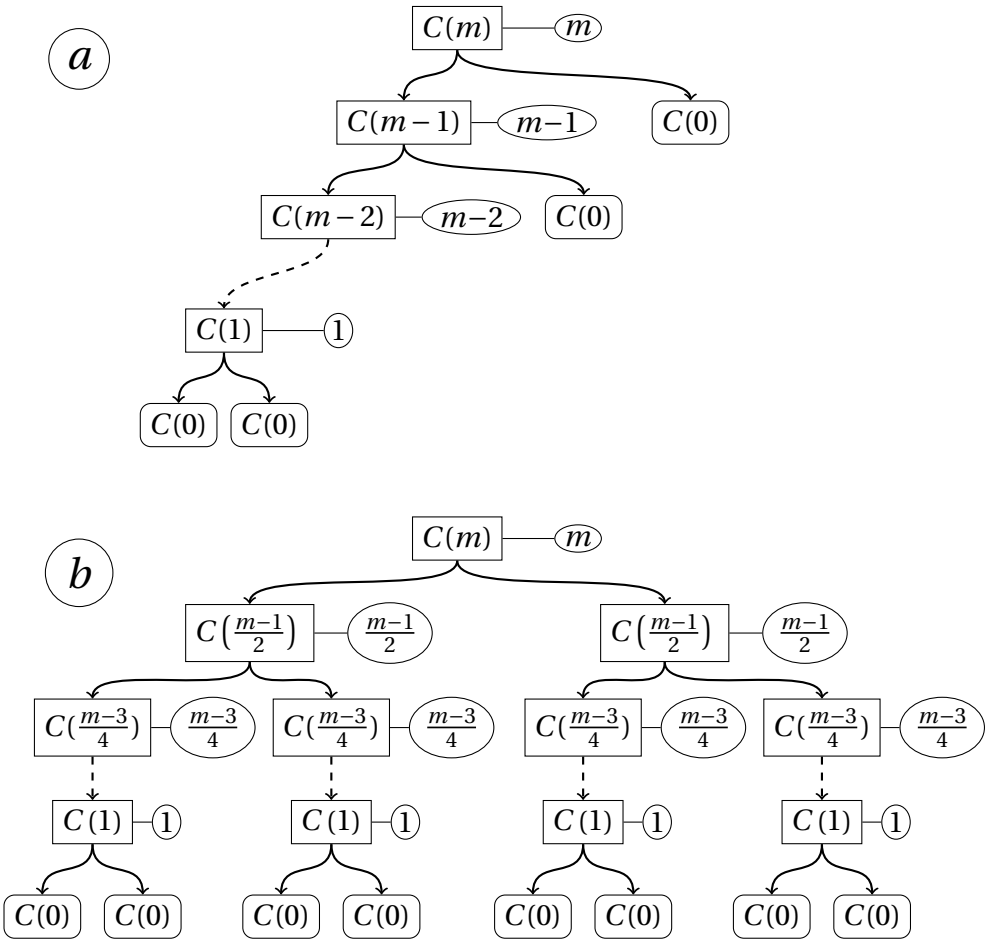


Figure 6.2: Evaluation trees of the CR $C(m)$.

like behaviour, which usually do not admit a linear UBF. Again, we first assume that the non-zero cost expression in the atomic CR is of the form $\mathsf{nat}(l)$, then, at the end of this section, we describe how to handle other basic cost expressions. We start with a motivating example that is used throughout this section.

Example 6.5. Consider a CR C that is defined by the following equations

$$\begin{array}{lll} C(m) = & 0 & \{m \geq 0\} \\ C(m) = & \mathsf{nat}(m) + C(m_1) + C(m_2) & \{m = m_1 + m_2 + 1, m_1 \geq 0, m_2 \geq 0\} \end{array}$$

Note that the cost contributed by the recursive equation, in each application, is equal to the input m, and that its recursive calls are applied on m_1 and m_2 such that $m = m_1 + m_2 + 1$. This models a typical divide and conquer behaviour.

Depending on the values chosen for m_1 and m_2 in each application of the recursive equation, a call $C(m)$ can be evaluated to (asymptotically) different

values. For example, choosing $m_1 = m-1$ and $m_2 = 0$ in all applications results in the worst-case $\frac{m^2+m}{2}$, as depicted in Tree (a) of Figure 6.2. On the other hand, splitting $m-1$ equally between m_1 and m_2 results in the best-case which is in $\Theta(m \log_2 m)$, as depicted in Tree (b) of Figure 6.2. Applying PUBS on this example infers the AUBF $C^+(m) = \mathsf{nat}(m) * 2^{\mathsf{nat}(m)}$, which is imprecise. ■

Handling the CR of the above example using the tree-sum method requires a polynomial UBF template, otherwise it fails. This is because it has an evaluation tree with polynomial cost as Tree (a) of Figure 6.2. However, recall that we are interested in using only linear UBF templates, which then allows solving the corresponding VC efficiently. Thus, using the tree-sum method is ruled out. Instead, in what follows, we describe an alternative approach for handling such CRs that it is based on a different kind of VCs and uses only linear UBF templates.

The intuition behind our approach is that in divide and conquer examples, a cost expression $w(\bar{x}) * h(\bar{x})$ is a UBF for the corresponding CR if (I) $w(\bar{x})$ is a *cost expression* that bounds the total cost contributed by *each level* of any evaluation tree; and (II) $h(\bar{x})$ is a *cost expression* that bounds the height of all evaluation trees. For example, for the CR of Example 6.5 we have $w(m) = \mathsf{nat}(m)$ and $h(m) = \mathsf{nat}(m)$, which results in the UBF $C^+(m) = \mathsf{nat}(m)^2$. Thus, we could aim first at inferring $w(\bar{x})$ and $h(\bar{x})$, and then combine them into a UBF for the corresponding CR. The advantage is that these bounds are typically linear, and thus can be synthesized efficiently. The next example describes the basics of this approach, by deriving appropriate VCs and synthesizing $w(m)$ and $h(m)$ for the CR of Example 6.5.

Example 6.6. Consider the CR of Example 6.5 and, for simplifying the presentation, let φ_2 refer to the constraints attached to the second equation. We say that a given cost expression $h(m)$ is a valid bound on the height of any evaluation tree of C if the following VC is valid:

$$\forall m, m_1, m_2 : \varphi_2 \quad \rightarrow \quad h(m) \geq 1 + h(m_1) \wedge h(m) \geq 1 + h(m_2)$$

Intuitively, since the height in this case corresponds to the number of consecutive applications of the recursive equation, the above VC requires that $h(m)$ covers one application of the recursive equation, and further ones that might arise through each recursive call. Note that each recursive call is considered separately (the clause has a conjunction inside), since they correspond to different paths in the evaluation tree. Observe that it is crucial that $h(m)$ is a cost expression, since this guarantees that it is always non-negative, otherwise the above VC is not correct.

Similarly, we say that a cost expression $w(m)$ is a valid bound on the cost of each level if the VC defined as a conjunction of the following clauses is valid:

$$\begin{array}{lcl} \forall m, m_1, m_2 : \varphi_2 & \rightarrow & w(m) \geq \mathsf{nat}(m) \\ \forall m, m_1, m_2 : \varphi_2 & \rightarrow & w(m) \geq w(m_1) + w(m_2) \end{array}$$

Intuitively, the first clause requires that $w(m)$ covers the cost contributed by the recursive equation at any level and the second clause requires that it also covers the next level. Again, it is crucial that $w(m)$ is a cost expression, since this guarantees that it is always non-negative.

In order to automatically synthesize the cost expressions $h(m)$ and $w(m)$ for which the above VCs are valid, we use the UBF templates $h(m) = \mathsf{nat}(h_{\mathrm{m}} m + h_{\mathrm{c}})$ and $w(m) = \mathsf{nat}(w_{\mathrm{m}} m + w_{\mathrm{c}})$, and then QE to eliminate the universally quantified variables. This results in the constraints $h_{\mathrm{m}} \geq 1 \wedge h_{\mathrm{c}} + h_{\mathrm{m}} \geq 1$ and $w_{\mathrm{m}} \geq 1 \wedge w_{\mathrm{m}} \geq w_{\mathrm{c}} \wedge w_{\mathrm{m}} + w_{\mathrm{c}} \geq 1$, that admit the solutions $h_{\mathrm{m}} = 1, h_{\mathrm{c}} = 0$ and $w_{\mathrm{m}} = 1, w_{\mathrm{c}} = 0$, respectively. Thus, $h(m) = \mathsf{nat}(m)$ and $w(m) = \mathsf{nat}(m)$ are valid bounds on the height and the cost of each level. ■

For a general atomic CR, unlike the CR that we used above, there might be recursive equations that do not contribute to the cost, i.e., they have local cost zero. This means that some level of the evaluation tree will have cost 0, and it would be more precise to ignore these levels when computing $h(\bar{x})$. To achieve this, instead of bounding the height of an evaluation tree, we bound the number of nodes with non-zero cost in each path from the root to a leaf. Similarly, instead of considering the level of a node as its distance from the root, it would be more precise to define it as the number of non-zero nodes from the root to that node. This generalization is formalized in the article.

Handling Arbitrary Basic Cost Expressions. Now we discuss how the above technique is extended to handle an arbitrary basic cost expression b, i.e., $b = m$, $b = m^{\mathsf{nat}(l)} - 1$, or $b = \log_m(\mathsf{nat}(l) + 1)$, instead of only $b = \mathsf{nat}(l)$. Note, however, that we still require that only one equation has a non-zero basic cost expression. Recall that in the level-sum method we have two VCs, the one that corresponds to $h(\bar{x})$ remains the same since the height of an evaluation trees does not depend on b. Handling the VC that corresponds to $w(\bar{x})$ is done exactly as we described for the case of the tree-sum method.

3 Handling General CRSs

Without loss of generality, we assume that the input CRS is a stand-alone CR, i.e., it includes only one CR symbol C. Handling the general case is done exactly as in PUBS, by solving them one CR at a time as in Section 2.5.

The tree-sum and level-sum methods were designed to handle atomic CRs in which all equations contribute zero to the cost, except one equation that can contribute a basic cost expression. However, in practice, CRs have several equations with non-zero cost, and, moreover, they use arbitrary cost expressions not necessarily basic. In what follows we describe our approach for handling such general CRs. It is based on the idea of breaking the input CR into several atomic CRs, solving the atomic CRs into UBFs by using the tree-sum or level-sum methods, and finally combining these UBFs into a UBF for the original CR.

For simplicity, we may assume that cost expressions in the input CR do not use the max operator, for if they use it, we can replace each max operator by the sum of its parameters, just as we did in Section 3.2. Another alternative is to clone the corresponding equation into several cases, one for each parameter of the max operator. We also assume that cost expressions are given in normal form, namely, each cost expression is a sum of products of basic cost expressions. We present our approach in two steps: We first extend atomic CRs to allow using a product of basic cost expressions; and then we consider a general case in which we have several equations that contribute arbitrary non-zero cost expressions (given in a normal form).

Products. Assume a given CR C in which all equations contribute zero, except for one equation that contributes $e = b_1 * \cdots * b_n$ where b_i, $1 \leq i \leq n$, are basic cost expressions. Our method to solve C into a UBF proceeds as follows:

1. We pick b_i, for some $1 \leq i \leq n$, and replace e by this b_i. This results in a CR E that is atomic;
2. We solve E into a UBF $E^+(\bar{x})$ using the tree-sum or level-sum methods;
3. For any b_j, with $j \neq i$, we compute its maximisation, denoted by $\hat{b}_j$, as described in Section 2.5; and
4. We define $C^+(\bar{x}) = \hat{b}_1 * \cdots * \hat{b}_{i-1} * E^+(\bar{x}) * \hat{b}_{i+1} * \cdots * \hat{b}_n$ as a UBF for C.

The correctness of this method relies on the fact that $\hat{b}_j$ is larger than any instance of b_j, and, moreover, it is a constant expression since it is given in terms of the initial input parameters and not in terms of the CR variables. Therefore we can factorized it out of the CR. Note that the nondeterministic choice of i in the first step might lead to different UBFs.

Example 6.7. Consider a CR C that is defined by the following equations:

$$\begin{array}{ll} C(m,p) = 0 & ,\{m = 0\} \\ C(m,p) = \mathsf{nat}(q) * \mathsf{nat}(n) + C(m-n,p') & ,\{m \geq n \geq 1, p \geq q, p \geq p' \geq 0\} \end{array}$$

Note that the cost contributed by the second equation is a product of two basic cost expressions. We replace this product by $\mathsf{nat}(n)$, which results in the following atomic CR:

$$\begin{array}{ll} E(m,p) = 0 & ,\{m = 0\} \\ E(m,p) = \mathsf{nat}(n) + E(m-n,p') & ,\{m \geq n \geq 1, p \geq q, p \geq p' \geq 0\} \end{array}$$

Using the tree-sum method we solve E into the UBF $E^+(m,p) = \mathsf{nat}(m)$. Now we maximise the other basic cost expression $\mathsf{nat}(q)$ into $\mathsf{nat}(p)$, which is the maximum value to which $\mathsf{nat}(q)$ can be evaluated. Note that p here refers to the initial input value and not to the parameter of the CR E. Finally, we define $C^+(m,p) = \mathsf{nat}(p) * \mathsf{nat}(m)$ as a UBF for CR C. ■

The General Case. Assume a general CR C with k equations, such that the i-th equation contributes a cost expression $e_i = P_{i1} + \cdots + P_{in_i}$, where each addend P_{ij} is a product of basic cost expressions. Our method to solve C into a UBF proceeds as follows:

1. For each product P_{ij}, we define a CR E_{ij} that is derived from C simply by removing all other products from C;
2. We solve each E_{ij} into a UBF $E^+_{ij}(\bar{x})$, as above; and
3. We build the UBF $C^+(\bar{x})$ of $C(\bar{x})$ as $C^+(\bar{x}) = \sum_{i=1}^{k} \sum_{j=1}^{n_i} E^+_{ij}(\bar{x})$.

The correctness of this approach relies on the fact that each CR E_{ij} models the maximum possible contribution of the addend P_{ij} to the total cost of C.

Example 6.8. Consider a CR C that is defined by the following equations:

$$\begin{array}{lll} \text{Eq.1} & C(m,n) = 0 & ,\varphi_1 \\ \text{Eq.2} & C(m,n) = \underbrace{\mathsf{nat}(n)+}_{P_{21}} \quad C(m,n_1) & ,\varphi_2 \\ \text{Eq.3} & C(m,n) = \underbrace{\mathsf{nat}(m_0)}_{P_{31}} + \underbrace{\mathsf{nat}(n)}_{P_{32}} + C(m_1,n_1) + C(m_2,n_2) & ,\varphi_3 \end{array}$$

where $\varphi_1 \equiv \{m_0 = 0, n = 0\}$, $\varphi_2 \equiv \{n \geq 1, n = 1 + n_1\}$, and $\varphi_3 \equiv \{m \geq m_0 + m_1 + m_2, n \geq 1 + n_1 + n_2\}$. In this CR there are three products: $P_{21} = \mathsf{nat}(n)$, $P_{31} = \mathsf{nat}(m_0)$, and $P_{32} = \mathsf{nat}(n)$.

To solve the CR C, we first split it into the CRs E_{21}, E_{31}, and E_{32}, depicted in Figure 6.3. Solving these CRs we get the UBFs $E^+_{21}(m,n) = \mathsf{nat}(n)^2$, $E^+_{31}(m,n) = \mathsf{nat}(m)$, and $E^+_{32}(m,n) = \mathsf{nat}(n)^2$. Then, we define $C^+(m,n) = \mathsf{nat}(n)^2 + \mathsf{nat}(m) + \mathsf{nat}(n)^2$ as a UBF for $C(m,n)$. ■

$$
\begin{array}{lll}
E_{21}(m,n) = & 0 & ,\varphi_1 \\
E_{21}(m,n) = & \mathsf{nat}(n)+ \quad E_{21}(m,n_1) & ,\varphi_2 \\
E_{21}(m,n) = & 0+ \quad E_{21}(m_1,n_1)+E_{21}(m_2,n_2) & ,\varphi_3 \\
E_{31}(m,n) = & 0 & ,\varphi_1 \\
E_{31}(m,n) = & 0+ \quad E_{31}(m,n_1) & ,\varphi_2 \\
E_{31}(m,n) = & \mathsf{nat}(m_0)+ \quad E_{31}(m_1,n_1)+E_{31}(m_2,n_2) & ,\varphi_3 \\
E_{32}(m,n) = & 0 & ,\varphi_1 \\
E_{32}(m,n) = & 0+ \quad E_{32}(m,n_1) & ,\varphi_2 \\
E_{32}(m,n) = & \mathsf{nat}(n)+ \quad E_{32}(m_1,n_1)+E_{32}(m_2,n_2) & ,\varphi_3
\end{array}
$$

Figure 6.3: Decomposition of a CR in three smaller CRs in the general case

4 Experimental Evalauation

We implemented our techniques as an extension of PUBS. The implementation took less than 750 lines of PROLOG code.

Precision. To evaluate the asymptotic precision of our techniques, we gathered from related literature some programs. Although this programs are short, they pose to cost analysis some challenge that is solved by our methods. In particular, for the tree-sum method we consider some programs usually described using the notions of amortised cost, such as the procedure to add many elements into a list implemented with a resizable array [45, §17], the procedure to add and remove many elements from a queue implemented with two stacks [103]. For the level-sum method we consider some *divide and conquer* algorithms, like QuickSort. In the article we show for each example the AUBF obtained by COSTA using the method to solve CRs of [5], and the one obtained using ours. In all examples, the methods of [5] obtain an imprecise AUBF, while our method obtains the precise AUBF of the program. Although PUBS already had a technique for solving divide-and-conquer CRs [5], it had a restricted applicability. For instance, it can not be applied to bind the cost of QuickSort.

Scalability Following [5, §10], we have composed the programs of the previous experiments to build a set of programs of increasing size as set of benchmarks. For each benchmark, we measured the time for solving the CRS using the methods of PUBS and using ours. In the experiments, we observed that the runtimes of our solving method is reasonable, and within a factor with respect to the runtime of the methods of [5].

We have also compared our approach to the ACRP prototype of [18]. For all benchmarks, ACRP failed to obtain an UBF in less than a minute. This is ex-

pected since ACRP is based on a general QE procedure for nonlinear real arithmetic, which is not scalable. Instead, our methods solved all programs in a few seconds. This is due to the use of a QE method based in linear programming.

In short, in our experiments we see that our method is more precise than the method of [10], and that its scalability is similar to that of previous PUBS methods.

5 Related Work

Automatic Solvers for Recurrence Relations. The first support for automatically solving RRs into closed form appeared in computer algebra systems like MACSYMA [78] and MATHLAB [42]. PURRS [27] is a relatively recent system that aims at solving recurrence relations as well, however, it tries to approximate the solution when it fails to find an exact one. All these systems assume deterministic recurrences, which cannot directly model the cost of nondeterministic (abstract) programs.

PUBS. The closest works to ours are those developed in the context of PUBS, which solves CRSs into UBFs and LBFs. The syntax, semantics of CRSs, and differences from recurrence relations, are described in [5]. This article also discusses two methods for solving CRSs: (I) the node-count method, that we described in Section 2.5; and (II) the level-count method, which aims at solving CRSs that originate from divide and conquer algorithms as our level-sum method, but it has restricted applicability.

Cost expressions were also introduced in [5], as a form of monotonic expressions that are generated from linear expressions. The key insight in cost expressions is that, since they are monotonic on their nat components, maximising a cost expression is done by maximising the linear expressions inside the nat expression. Thus, cost expressions are a leverage to reason about complex expressions using linear programming techniques. In the same way, our decomposition of a CR into atomic CRs is a leverage to solve complex CRs using QE tools for linear arithmetic.

PUBS also relies on the techniques of [14] which solve CRSs by first transforming then into RRs that approximate their worst-case, and then solve these RRs with computer algebra systems. This approach can obtain UBFs that are more precise than those of [5], and can be easily used to infer LBFs as well. However, its applicability is more restricted, and still obtains an asymptotically imprecise UBF for our examples.

Logical Reasoning and Quantifier Elimination. The tree-sum method and the level-sum method are inspired by the inductive assertion method for proving program correctness [34]. The use of QE methods for linear arithmetic has been used in other related works. It is the basis for automatic inference of linear ranking functions [26], and lexicographical ranking functions [15]. In [124] inductive assertions, combined with templates, are used for synthesis and verification of programs. Their tool handles a wider class of template predicates, that include disjunctions and conjunctions of linear constraints.

Reachability Bound. Our level-sum method bounds the height of the evaluation trees of a CRS, which can be seen as the number of *subsequent* visits to an equation, which is a *reachability bound*. Such a bound can be directly computed as a linear ranking function [26]. In systems with bounded nondeterminism, it can also be computed from a lexicographical (or multidimensional) linear ranking function [15]. In [66], the authors define this problem and present some tools for computing such bounds using an SMT solver. Their technique allows computing UBFs that are products or a max of two nat expressions. In [140], they combine this approach with the size-change abstraction. Some works have proposed inferring polynomial ranking functions: some by using QE methods for nonlinear real arithmetic [38], others by using the technique of Lagrangian relaxation to reduce it to a linear programming problem [47]. Other techniques have been proposed in [97].

6 Further Reading

In this Chapter we have described the technical contributions of [17]. We explained how to solve a CR by splitting it into several atomic CRs, how to solve these atomic CRs into corresponding UBFs, and how to combine these UBFs into a UBF for the original CR. We have also explained the tree-sum and level-sum methods for solving atomic CRs. These methods are based on the use of universally quantified FOL formulas as in Chapter 5, however, they can be solved efficiently using QE for linear arithmetic.

The article extends and formalises the content of this chapter. It includes a formal definition for the syntactic class of atomic CRs. It also formalises the tree-sum method and the level-sum methods, and provides correctness statements and their corresponding proofs. The article describes the QE procedure that we use to solve the different VCs, which is based on the use of Farkas Lemma, and how to adapt it to handle the nat expressions in the VCs. The article also contains the tables with the results of our experiments.

7 | Conclusions, and Future Work

In this chapter we draw the conclussions of this dissertation. In Section 7.1 we describe the achievements, impact, and practical applicability of our contributions. In Section 7.2 we give a extended discussion about our conclussions about the relation between the classical and the amortised approaches to cost analysis. Finally, in Section 7.3 we sketch some possible lines for future work.

1 Objectives, Achievements, and Impact

Our first objective was to adapt existing tools to compute asymptotic UBFs as well, which are a natural choice for specifying the expected cost of a program under development since they are (I) less sensitive to small changes in the program, and (II) concise and easier to interpret for large programs. This objective has been achieved in Chapter 3 as follows:

- In a first step, we have followed a transformational approach and developed automatic methods to transform UBFs, or more precisely cost expressions, to asymptotic form. The advantage of this approach is that it can be applied directly to UBFs obtained by any cost analyser, not necessarily COSTA. The disadvantage is that we still have to obtain non-asymptotic UBFs before applying the transformation.
- In a second step, we have developed a novel asymptotic CRS solver that directly solves CRSs into asymptotic UBFs. This was achieved by interleaving the different phases of PUBS with the transformation of cost expressions to asymptotic form. This solver can be used for any cost analysis that generates CRSs. Experimental evaluation confirmed the superiority of this last approach over the first one.

An important feature of our approach, unlike previous ones [126], is that it takes context information into account.

Our second objective was to explore the precision gap between existing approaches to cost analysis. This included (I) understanding the reasons for which COSTA, and Wegbreit's classical approach in general, infers imprecise UBFs for some programming patterns; and (II) developing novel techniques to close this precision gap. We have identified several sources for this imprecision, each one is related to a different phase of COSTA, and proposed novel techniques to overcome these problems:

- In Chapter 4 we have addressed the imprecision induced by the use of non-linear arithmetic operations, which cannot be modeled in COSTA since it

relies on the use of linear constraints. We have observed that in the presence of context information, one can model nonlinear operations with linear constraints only, and, moreover, this can be encoded directly in the ACR program which avoids significant changes to the underlying analyser.
- In Chapter 5 we have observed a surprising relation between the output of a procedure and its cost, which is the main reason for some well-known precision problems. This imprecision is related to the phase that transforms the ACR program to the CRS (since output parameters are removed). To solve this problem, we have developed novel techniques that are based on specifying the cost as a FOL formula over some UBF templates, and then using QE procedures and SMT solving to compute concrete UBFs. Our approach supports releasing resources as well, and thus gave rise to the notion of peak-cost.
- In Chapter 6, we combined the techniques developed in Chapter 5 with those of PUBS, to develop a precise and scalable method to solve a CR into a UBF. This was achieved by splitting the input CRS into several atomic ones, whose cost can be modeled using simpler FOL formulas that can be solved by an efficient QE procedure. Our experiments show that our new techniques are close to those of Chapter 5 in terms of precision, and to those of PUBS in terms of scalability.

Concluding the achievements related to the last objectives: apart from justifying, for the first time, the source of some imprecision problems in the classical approach to cost analysis, our major contribution is to apply *The Calculus of Computation* [34] to static cost analysis, by turning UBF inference and CRS solving into the verification of an abstract program using the current progress on SMT solving [89] and QE [81] procedures.

Our contributions have a fundamental impact on both practical and theoretical aspects of cost analysis. In the practical case, this is witnessed by the new features that were added to COSTA, and the improvements in its applicability, precision, and scalability.

- Our first contribution extends the features of COSTA with the possibility of inferring asymptotic bounds. As shown in Section 3.5, the asymptotic CRS resolution improves the scalability of COSTA and, indirectly, its practical applicability to the larger programs.
- Our second contribution has extended the applicability of COSTA to JBC programs whose cost depends on a nonlinear instruction. As shown in Section 4.2, this contribution allows COSTA to infer a stronger value relations, with which PUBS is able to infer more precise AUBFs. Moreover, our solution does not compromises the scalability of COSTA, not only because

uses a non-disjunctive abstract domains, but also because it does not increase the size of the generated CRS.

- The impact of our third contribution (Chapter 5) goes beyond the practical aspects: it sheds new light on the relation between the different approaches to cost analysis. On the practical side, as discussed in Section 5.5, we implemented our approach in a prototype called ACRP. ACRP is able to infer precise AUBFs for many programs for which COSTA fails, including some classical examples related to amortised analysis, and some nested loops. However, ACRP uses the SMT and QE methods for the FOL theory of real fields (real numbers), which are not scalable. As a result, this contribution has no practical applicability in the context of COSTA, which is why we have not integrated it in there.
- In our fourth contribution we have improved the precision of PUBS, by developing a new method to solve a CR into a UBF. As shown in Section 6.4, our technique achieves a better asymptotic precision than those of [5, 14]. By relying on a QE method for linear real arithmetic, our method does not damage the scalability of COSTA. Our method also extend the applicability of COSTA and PUBS to some programs for which there are no linear ranking functions, even to some non-terminating programs, since our method does not need to compute a linear ranking function.

Concluding the impact, apart from the clear evidence (see Sections 3.5, 4.2, 5.5, and 6.4) on the practical improvements to COSTA, we believe that the theoretical contributions of Chapter 5 will make it easier to transfer knowledge and implementation techniques from one approach of cost analysis to another.

Applicability of our Contributions

Our major contributions are developed for ACR programs and CRSs. Thus, they are generic in the source programming language, the cost model, or the size measure used. Therefore, they can be also applied to non-accumulative cost models like heap consumption [13], or concurrent tasks [11], or to any user-definable cost model [99, 98]. In the same way, they can be applied if the size measure is the path-length of heap-allocated data structures [123], or the introspective size measures for functional data structures used in RAML [73], or any user definable size measure [65]. With respect to programming languages, the abstract compilation of the RBR into the ACR can be adapted to consider such features as field-sensitive analysis over heap-allocated data [6]. Our contributions can also be applied to the COSTABS [138] analyser for the concurrent ABS language.

2 Amortised Cost Analysis

When we started our research, we found several programs for which COSTA inferred imprecise UBFs. Some of these were (functional) programs for which the methods for automated amortised analysis [79, 73] could infer precise UBFs. Some other programs consisted of a single loop for which COSTA inferred an imprecise UBF, while the methods of [64, 66] could infer the precise *amortised* complexity. Finally, there were some data structures for imperative languages [45, §17], that COSTA could not handle. Since all these examples were described in terms of amortised cost or amortised complexity, while seeking for the source of imprecision in COSTA we had amortised analysis in mind.

Unfortunately, we did not found a clear definition of amortised cost in the literature. The seminal paper of Tarjan [129] described two methods for manual cost analysis, the banker's method and the physicist's method, which he used to analyse programs that performed sequences of operations on some data-structures. He used the word *amortisation* to illustrate the change of focus from analysing one operation to analysing a sequence of them. Thus, the term *amortised cost* is a metaphor to describe the result of studying the *worst-case* cost of a program with certain methods. However, other works [103] use it to indicate a different property of program semantics.

We believe that our contributions help to clarify the notion of amortised cost, and exactly identify the components due to which COSTA was imprecise for the programs mentioned above:

- On one hand, the output-cost codependency described in Section 5.1 explains why COSTA infers an asymptotically imprecise UBF, for instance, for the nested loops in [64, 66], as well as in the classical binary counter data structure [45, §17], or in Tarjan's stack example [129].
- On the other hand, there are programs associated to the notion of amortised cost for which the output-cost codependency does not appear. In terms of COSTA, this means that corresponding CRSs accurately model the corresponding costs. This is the case of the Dynamic Array data structure [45, §17.4], or the implementation of a functional queue with two stacks [103, §3]. For such programs, the imprecision is related to the CRS resolution phase, in particular to the inability of PUBS to track dependencies between the costs of the different iterations of a loop, as in Chapter 6.

Our work is the first to make the above observations.

3 Future Work

There are several ways in which our contributions can be improved:

- In Chapter 6 we focused on improving the asymptotic precision of the computed UBFs, and improving the asymptotic one is left for future work. For this, we need a better way to handle max expressions, or a CR with many equations. For instance, we may use a greedy counter optimisation technique in the manner of SPEED [65].
- The techniques of Chapter 6 for solving atomic CRs can be extended to handle more CRs. This can be done by looking for patterns of evaluation trees that are typically used in complexity analysis [45].
- A we have discussed in Chapter 5, it easy to adapt our techniques to infer LBFs on the net-cost or peak-cost of terminating programs. As future direction could address this problem for non-terminating programs.
- The techniques of Chapter 5 required UBF templates to be provided by the user. A future research direction could concentrate on inferring such templates automatically, by syntactic analysis of the corresponding CRS.
- The use of QE and SMT solvers over linear arithmetic can be adapted for other problems in the context of solving CRSs. For example, we can use in the context of [14] to infer the arithmetic or geometric increasing/decreasing factors required for translating CRSs to RRs. We could also use it for building an alternative maximisation procedure that is not based on invariant generation, or for inferring nonlinear ranking functions.

All these improvements would have an impact both on the precision and on the applicability of cost analysis.

Parte II

Resumen de la Investigación

8 | Introducción

Toda máquina tiene una funcionalidad para la cual se construye y un entorno con el que interactúa. Un **recurso** es cualquier entidad del entorno que la máquina usa. Para usar una máquina se debe proveer su entorno con los recursos que necesita, y para ello es esencial saber por adelantado la cantidad de éstos.

En el caso particular de los programas de ordenador, su funcionalidad es procesar unos datos de entrada y generar unos de salida; su entorno de ejecución incluye, entre otros, los componentes del ordenador en que se ejecuta, tales como como la memoria, procesadores, etc. En este contexto, un **recurso** puede ser una magnitud física como el tiempo o la energía, un elemento lógico del hardware como un ciclo de CPU, una palabra en memoria, o un paquete de red; puede ser una unidad lógica de software, como un objeto, una llamada a un método, o un archivo; puede ser incluso el dinero que el usuario ha de pagar por una operación facturable como, por ejemplo, enviar un mensaje de texto desde un teléfono móvil. Cada recurso se **consume** de manera distinta: el tiempo pasa mientras el programa se ejecuta, las celdas de memoria se ocupan temporalmente, las llamadas a un método ocurren, etc.

Para estimar el **consumo de recursos** (o **coste**) de un programa, hay dos enfoques principales: el dinámico y el estático. El enfoque dinámico o experimental usa simuladores para ejecutar el programa con varios datos de entrada, y mide cuántos recursos se consumen. Este método es fácil de realizar, y da resultados exactos para la entrada escogida; sin embargo, un gran inconveniente es que es incompleto, ya que los resultados solo valen para las entradas escogidas. En cambio, en el enfoque estático o analítico sí se puede obtener información sobre el consumo de recursos de todas las ejecuciones posibles del programa, sin ejecutarlo ni una sola vez. Este enfoque se basa en esta idea:

> "La programación es una ciencia exacta en tanto que, en principio, todas las consecuencias de ejecutar [un programa] en un entorno dado se pueden descubrir a partir del texto del programa." [72]

El proceso de descubrir esas consecuencias "a partir del mismo texto del programa" sin necesidad de ejecutarlo, se conoce como **análisis estático**.

La ventaja de este enfoque es que, al dar información sobre todas las ejecuciones, un buen análisis basta para verificar que un programa siempre funciona como se desea. El inconveniente es que esto es demasiado difícil y caro, ya que analizar a mano un programa de tamaño medio no es asequible. Para hacer un análisis estático se necesita un dominio de las matemáticas que solo unos pocos especialistas bien entrenados poseen, e incluso para éstos, analizar un

programa sencillo a mano sería un proceso aburrido, largo, y propenso a error. Para que el análisis estático de programas sea realmente útil, debe realizarse automáticamente, esto es, por ordenador.

1 Análisis Estático de Coste

El análisis estático automático [100] estudia cómo construir un programa, llamado analizador estático, que toma como entrada el código de una aplicación (en algún lenguaje) y calcula información sobre su comportamiento en ejecución. Esta tesis se encuadra dentro del campo del **análisis estático automático de coste**, que estudia cómo construir un programa, llamado **analizador de coste**, capaz de recibir como entrada una aplicación y calcular [1] cuántos recursos consume en ejecución. Un analizador de coste suele ser paramétrico en el recurso cuyo consumo se analiza, para lo que se usan **modelos de coste**, que son funciones que asocian un coste a cada instrucción de un programa.

La salida de un analizador de coste es una **función de coste** que asocia cada ejecución del programa con su coste. Debido a la indecibilidad de los problemas subyacentes, a menudo dicha función no calcula el coste exacto del programa, sino una **función de cota superior** (*upper bound function* o UBF) o una **función de cota inferior** (*lower bound function* o LBF) para el coste. [2]. Como es usual en el campo del análisis de algoritmos [45, 118], estas funciones se definen sobre varias variables numéricas y vienen dadas en **forma cerrada**, esto es, son deterministas y no recursivas. Para manejar datos de entrada no numéricos, los analizadores de coste utilizan **medidas de tamaño** que asignan a cada instancia de una estructura de datos un valor numérico, como por ejemplo la longitud de una lista, la longitud de un array, etc.

Cuando se usan estas medidas de tamaño, la función de coste suele ser una UBF para el coste en el caso peor o una LBF para el coste en el caso mejor. Ello se debe a que la abstracción de tamaño puede asociar el mismo valor abstracto a dos datos de entrada distintos; y también al indeterminismo del lenguaje de programación que se analiza, por el que un programa puede tener costes diferentes para un mismo valor de entrada, según la elección indeterminista.

En el campo del análisis de algoritmos, una preocupación es la escalabilidad del coste de un algoritmo o programa con respecto al tamaño de su entrada. Para responder a tales cuestiones, se suelen usar las notaciones asintóticas *Big O*, *Big Omega* y *Big Theta* [88]. Estas notaciones indican intuitivamente que, a partir de algún tamaño de la entrada, una función es menor, mayor o proporcional a otra. Por ello, es deseable que las funciones de coste se den en **forma**

[1] Dentro de los límites de la teoría de la computación.

[2] Se usan las palabras **función de cota**, o **cota**, para referirnos a una UBF o una LBF.

asintótica, además o en lugar de no asintótica.

Debido a la incompletitud inherente del análisis estático, a veces un analizador de coste no puede calcular una función de cota, aunque exista. Las propiedades clave de un analizador de coste son pues su aplicabilidad, precisión, y escalabilidad. En concreto:

- La **aplicabilidad** es la clase de programas para los que el analizador puede inferir cotas no triviales.
- Para tales programas, la **precisión** es la relación entre la cota que el analizador calcula y el coste en el caso peor (o mejor).
- La **escalabilidad** es la habilidad del analizador para mantener el nivel de rendimiento al aumentar el tamaño de los programas que se analizan.

La aplicabilidad puede estar restringida por características del lenguaje, como el acceso a campos [6]; por propiedades semánticas, como la terminación [5, 79, 140]; o por las clases de complejidad de las cotas, por ejemplo a cotas lineales [79, 35] o polinómicas [65, 84, 73]. Respecto a la precisión de las cotas, si bien algunos trabajos se centran en mejorar la precisión funcional [73], el objetivo prioritario es mejorar la precisión *asintótica*, esto es inferir cotas que estén en el orden asintótico exacto del coste del programa.

Aunque hay otros enfoques para el análisis de coste automático, los más relevantes para nosotros son el enfoque clásico y el de análisis amortizado, que se describen a continuación.

Enfoque Clásico.

El enfoque clásico para analizar manualmente un programa [45, §2.2] consiste en obtener del programa un sistema de ecuaciones que define su coste y posteriormente resolverlo a una función de cota. Por ejemplo, con un modelo de coste del número de pasos de ejecución, las instrucciones, secuencias, y ramificaciones, se representan con a constantes, sumas, y máximos, y los bucles se representan con relaciones de recurrencia [3] (abreviado como RR).

> "Las relaciones de recurrencia aparecen a menudo [...] por una correspondencia directa entre la representación recursiva de un programa y la representación recursiva de sus propiedades." [118]

Las RRs solo capturan las propiedades esenciales de un algoritmo que no varían entre las distintas implementaciones. Después de escribir el coste de un programa como un sistema de RRs, en la aproximación manual el siguiente paso es resolverlas a una solución en forma cerrada [61, 111, 58].

Basado en este enfoque, Cohen y Zuckerman [44] presentan un analizador para un lenguaje imperativo simple que es automático únicamente en la fase

[3] También conocidas como ecuaciones de recurrencia o ecuaciones en diferencias.

de generación de las RRs. El primer *análisis mecánico de programas* íntegro lo presentó Wegbreit [133]. Ahí se describe METRIC, un analizador de coste para LISP, que opera en dos fases: la primera transforma un programa LISP automáticamente a un sistema de RRs que describe su coste; la segunda usa un resolutor automático de RRs para obtener una solución en forma cerrada.

Siguiendo el método de Wegbreit, la aproximación clásica divide el análisis estático automático de coste en dos fases independientes: análisis y resolución.

1. La fase de **análisis** transforma el programa a un sistema de RRs. Esta fase se construye de manera diferente según el lenguaje de programación, el modelo de coste, o las medidas de tamaño usadas.
2. La fase de **resolución** del sistema de RRs calcula una solución en forma cerrada. Al ser las RRs una representación puramente numérica, esta fase es independiente del lenguaje de programación o del modelo de coste.

La ventaja del enfoque clásico es que, si bien la primera fase varía según el lenguaje, modelo de coste y medida de tamaño, la segunda fase (la que nos interesa) no depende de esos factores. Por lo tanto, un sistema de RRs se puede ver como un *programa abstracto*, que sirve para analizar cualquier lenguaje. Además, cuando se analizan dos programas que implementan el mismo algoritmo en lenguajes distintos se generan RRs (casi) idénticas. En este sentido, la primera fase sirve para filtrar, o *abstraer*, los detalles de implementación y dejar solo la esencia del algoritmo.

Después de Wegbreit, muchos trabajos han extendido o adaptado las dos fases del enfoque clásico. Puesto que estas fases son independientes, a continuación se comentan por separado los trabajos existentes para cada fase.

Generación de Relaciones de Recurrencia. La investigación en la primera fase, la de generar las RRs, se ocupa de manejar las características semánticas especiales de cada lenguaje, modelo de coste, o medida de tamaño. Respecto al lenguaje de programación, hay muchos trabajos que presentan análisis para lenguajes declarativos, ya que en estos lenguajes los programas coinciden directamente con su representación recursiva. Por tanto, hay analizadores para lenguajes funcionales [71, 132, 112, 62, 31, 131, 32, 51, 55], y lógicos [54, 99]. En cambio, para lenguajes imperativos u orientados a objetos la investigación ha sido menor [57, 98, 86]. COSTA [10] es el primer analizador que se desarrolló para programas orientados a objetos de Código Byte de JAVA (*Java Byte-Code* o JBC). Posteriormente, apareció el sistema COSTABS [3] que permite analizar programas concurrentes orientados a objetos escritos en el lenguaje ABS. Respecto a los modelos de coste, si bien muchos trabajos se centran en modelos de coste relacionados con el tiempo, también existen analizadores para modelos de coste como el espacio en memoria [13], número de tareas concurrentes [11], o consumo de energía [86]. Para la abstracción de tamaños, en el análisis de

lenguajes funcionales se emplean sistemas de tipos avanzados como los tipos dependientes [62] o tipos con tamaños [131, 119]. El tipo de función de coste que se infiere se centra habitualmente en UBFs para el coste en el caso peor, pero algunos trabajos también infieren LBFs para el coste en el caso mejor [95]. Incluso existen aproximaciones para calcular ambas [112]. Finalmente mencionamos algunos trabajos que aplican este enfoque a la inferencia del coste medio [71, 57]. En estos trabajos se tratan dos problemas difíciles: el primero es manejar las distribuciones de probabilidad en los datos, para lo que se requiere de una semántica especial [106]; el segundo es cómo tratar las propiedades combinatorias de algunas estructuras de datos [118, 58].

Relaciones de Recurrencia y Relaciones de Coste. Un problema que tienen los analizadores en esta fase, empezando por el de Wegbreit [133], es la distancia semántica entre las RRs deterministas y programas. En un programa puede haber caminos de ejecución que dependan de guardas o condiciones las cuales no se pueden representar en una RR. Como consecuencia, las ecuaciones que cubren caminos diferentes pueden superponerse (ser simultáneamente aplicables) para algunos valores del tamaño de las entradas. Para solucionar este problema existen distintas aproximaciones. Algunos autores [62, 51] usan directamente operadores de máximo, para mantener el determinismo. Karp propone usar RRs probabilísticas [82]. Basado en la teoría de Interpretación Abstracta [46], Rosendahl [112] propone considerar una RR no determinista como un *programa abstracto*, cuya semántica abstracta aproxima el conjunto de costes de las ejecuciones del programa. Esta idea también se usa en COSTA [10], donde a los RRs no deterministas se les denomina **sistema de relaciones de coste** (*Cost Relation System* o CRS). La ventaja de usar un CRS como un programa abstracto es que de un mismo programa abstracto se pueden inferir tanto las UBFs como las LBFs.

Resolución de Relaciones de Recurrencia. La investigación en esta fase se centra en desarrollar métodos para resolver automáticamente RRs, o bien un CRS, a una cota. A diferencia del intenso trabajo realizado en la fase anterior, la investigación en esta fase ha sido más lenta y menos amplia, quizá porque ya es de por sí difícil resolver a mano un sistema de RRs a cotas en forma cerrada. Cuando Wegbreit publicó su artículo, no existían dichos métodos. Fue su artículo el que inspiró los primeros métodos, desarrollados para sistemas de álgebra por computador, como MACSYMA [78], MATHLAB [42], o MAPLE [57, 113]. Muchos trabajos en este área se centran en cómo manejar funciones especiales, como polinomios o binomios [104, 105, 120]. Otros trabajos recientes son [36, 39]. El sistema PURRS [27] implementa algunos métodos para resolver o aproximar un sistema de RRs.

Sin embargo, todos estos métodos tratan RRs *deterministas*, no los CRSs que, por ejemplo, se usan en COSTA [10]. Para resolver CRSs o programas abstractos semejantes hay poco trabajo. Rosendahl [112] utiliza transformaciones de programas inspiradas en el sistema ACE [96]. El sistema PUBS [5] es el primero que resuelve una clase amplia de CRSs. Se basa en ver un CRS como un lenguaje de programación abstracto con una semántica operacional, y usa técnicas de análisis de programas para acotar el coste máximo. En particular, usa la conexión entre el cálculo de RR lineales y el análisis de terminación descrito en [83]. En [14] se describe una extensión de PUBS con la que se infieren UBFs más precisas, así como LBFs.

Análisis Amortizado

En los años 60 y 70, la aproximación clásica al análisis de coste se aplicó (manualmente) para obtener cotas precisas para muchos programas. Sin embargo, a principios de los 80 algunos investigadores vieron que, cuando se analizaba con este enfoque el rendimiento de algunos programas que operaban sobre ciertas estructuras de datos, como árboles auto balanceados, se obtenía una UBF imprecisa. Lo peculiar de esos programas era que cada operación individual podía ser cara (coste alto) o barata, según el estado de la estructura de datos. En un análisis según el enfoque clásico se considera que todas las operaciones en una secuencia son caras, lo que, si tal posibilidad no se da, lleva a una UBF asintóticamente imprecisa. Esto llevó a Robert Tarjan a desarrollar un enfoque alternativo al análisis de coste.

El análisis amortizado [129] es un enfoque para el análisis de coste que se centra en analizar no el coste de una operación de un tipo, sino el de una secuencia de tales operaciones. Este enfoque parte de la observación de que, aunque cada operación pueda ser cara o barata, en toda secuencia de operaciones hay suficientes operaciones baratas para *amortizar* entre éstas el coste de las caras. Para demostrar que se da dicha *amortización*, en este enfoque se usa la siguiente metáfora: se asume que la estructura de datos guarda alguna clase de ahorros, y se estudia qué relación hay entre el coste de cada operación y su efecto en los ahorros en la estructura de datos. Hay dos métodos principales para análisis amortizado de coste, que difieren en cómo se representan dichos ahorors: en el método del banquero, como créditos asociados a cada elemento de la estructura de datos; en el método del físico, como un potencial asociado a toda la estructura de datos. Estos métodos son equivalentes, así que se puede escoger entre uno u otro como más convenga. El enfoque de análisis amortizado es más preciso que el enfoque clásico, en tanto que obtiene una UBF asintóticamente precisa para la clase de programas antedicha.

Tras el artículo seminal de Tarjan, el análisis amortizado se ha usado para

analizar el coste de programas que operan en estructuras de datos en lenguajes tanto funcionales [103] como imperativos [45, §17]. Su uso para un análisis de coste, lo que se llama un *análisis amortizado automático*, lo introdujo Jost [75, 79]. Su trabajo considera un lenguaje funcional estricto de primer orden, y presenta un análisis, basado en sistemas de tipos, que infiere para cada procedimiento del programa, dos funciones de potencial: una sobre las entradas del procedimiento, la cual es una UBF en el coste del procedimiento, y otra función de potencial sobre la salida del procedimiento, que se usa para *pagar* el coste de las operaciones que se hacen sobre la salida.

El método de Jost se extendió posteriormente en varias direcciones. Campbell [35] estudia cómo inferir, para un lenguaje similar, una cota en el consumo de espacio de pila, lo cual requiere modificar el sistema de tipos. Rodriguez [109] aplica el análisis amortizado automático a un lenguaje orientado a objetos, para el cual hace falta tratar la herencia. El sistema RAML [73] es capaz de inferir UBFs que son polinomios multivariable, considerando un conjunto amplio de normas dirigidas por tipos, como por ejemplo el producto de polinomios sobre las longitudes de dos listas, o medidas más internas como la suma de las longitudes de la lista en una matriz. Jost et al. [80] tratan un lenguaje funcional de orden superior con evaluación estricta, cuyo análisis requiere tomar en cuenta el coste de evaluar cada aplicación parcial de una función. Simões et al. [122] consideran un lenguaje con evaluación perezosa, en el cual hay que manejar los *thunks*, esto es, llamadas a funciones cuya evaluación se suspende. Scherer y Hoffmann [117] consideran el caso en que el coste depende de los argumentos ya guardados en el cierre (*closure*) de una aplicación parcial de función. Hoffmann y Shao [74] extienden el análisis a un lenguaje imperativo, que incluye operaciones en arrays y de aritmética entera no lineal. Atkey [23] presenta un método para analizar el coste de programas sobre estructuras de datos enlazados, como las listas, inspirado en el método del contable y basado en lógica de separación [107].

2 Aplicaciones del Análisis de Coste

Existen diversas aplicaciones para el análisis de coste. En el desarrollo de hardware, se usa para analizar y verificar el tiempo de ejecución de los procesadores [137], sobre todo para sistemas en tiempo real; últimamente también interesa analizar el consumo de energía [86], sobre todo de sistemas integrados.

En el campo de la computación paralela, una posible aplicación se relaciona con el equilibrio de carga y el control de granularidad de los programas, esto es, en encontrar la manera óptima de descomponer un cálculo en varias tareas concurrentes y repartir éstas entre los procesadores. Como ejemplo, se puede

aplicar el análisis de coste para estimar la carga de trabajo adecuada para cada procesador [95]. Otra técnica más flexible es usar programas móviles autónomos (*autonomous mobile programs o AMP*) [55], donde cada AMP analiza el coste de las computaciones que tiene pendientes, y decide por sí solo si migra a otro procesador con menos trabajo. En una computación distribuida, dado que la red de comunicación entre los nodos puede suponer un cuello de botella, también importa saber cuántos mensajes se envían a través de la red [12]. Asimismo, si los programas se escriben en un lenguaje concurrente, es útil saber el número máximo de tareas concurrentes que pueden estar activas [11], ya que ése es el paralelismo máximo del programa. Otra posible aplicación se encuentra en el contexto de *cloud computing* [22], donde una empresa vende sus capacidades computacionales. En este contexto, el análisis de coste sirve para estimar, antes de una transacción, el precio de la computación que se compra.

Desde el punto de vista de un programador, el análisis de coste se puede utilizar para la verificación de aserciones que especifican el coste esperado de un programa. Por ejemplo, este mecanismo [94] está incorporado dentro del compilador del lenguaje CIAO [69]. En esta línea mencionamos también la certificación de consumo de recursos [50, 9], que está basada en la idea de anexar a un programa una demostración que certifique que cumple una determinada especificación. Este proceso forma parte de una técnica general conocida como la verificación constructiva de programas [135].

Liang [92] aplica el análisis de coste para la optimización de programas lógicos. También sirve para adaptar automáticamente un programa a la plataforma en que se ejecuta, por ejemplo, en el contexto del lenguaje y sistema PETABRICKS [20]. En éste, el programador escribe varios procedimientos, que pueden implementar algoritmos distintos, para realizar una misma tarea. El compilador y la plataforma de ejecución de este lenguaje escogen automáticamente cuál es el procedimiento más eficiente para cierta arquitectura o tamaño de datos. Es en este punto en que un analizador de coste puede ser útil.

3 Objetivos de la Tesis

La presente tesis se encuadra dentro del campo del análisis estático y automático de coste, usando el enfoque clásico. En este enfoque, el análisis distingue dos fases. La primera se encarga de generar programas abstractos, y la segunda de resolverlos a funciones de cota en forma cerrada. Un aspecto muy importante en este campo es conseguir que los analizadores gocen de la mayor precisión posible, sobre todo asintótica.

El objetivo principal de esta tesis es mejorar la precisión y aplicabilidad de los analizadores de coste basados en este enfoque, de forma que su precisión

sea similar a la de los analizadores basados en análisis amortizado, conservando una buena escalabilidad. En particular, nos interesa mejorar los técnicas de la segunda fase, para poder ir ***de programas abstractos a cotas asintóticas precisas en forma cerrada***. En síntesis, queremos construir [4] un analizador que infiera UBFs asintóticamente más precisas para una variedad más amplia de programas.

El reto principal es la diferencia de precisión entre el enfoque clásico y el enfoque de análisis amortizado. Se sabe que para algunos programas los analizadores basados en éste obtienen cotas más precisas que los basados en aquél, pero no hay una explicación clara de por qué sucede, ni un criterio para saber para qué programas. Este hecho conlleva a algunas concepciones erróneas sobre el enfoque de análisis amortizado, por lo que algunos textos [103] usan la noción de *coste amortizado* en contraposición al de coste *en el caso peor*, como si se trataran de propiedades distintas, en vez de métodos de análisis diferentes. Una concepción errónea es que el análisis amortizado de coste solo sirve para estructuras de datos, no para algoritmos o bucles. De hecho, los métodos de análisis amortizado automático [79] describen el potencial como una medida relacionada con los constructores de estructuras de datos.

Debido a dichas concepciones erróneas, durante mucho tiempo se pensaba que para poder inferir cotas precisas era necesario usar las técnicas del enfoque amortizado. La idea de inferir cotas precisas *sin* usar las técnicas del enfoque amortizado aparecen por primera vez en el contexto del proyecto SPEED [65]. La investigación en este proyecto se centra en el análisis de coste [5] de programas en un lenguaje imperativo, que consisten en bucles anidados en los que el bucle interno y externo comparten algunas variables de contador. Para estos programas, los analizadores basados en el enfoque clásico como COSTA [7, 1], así como otros analizadores que no siguen el enfoque clásico como [84], infieren una UBF imprecisa, o incluso fallan al inferir. En cambio, los trabajos del proyecto SPEED logran inferir automáticamente UBFs precisas usando técnicas relacionadas con el análisis de terminación de bucles [65, 64, 66, 140]. Esto sugiere que es posible lograr la precisión que logra un análisis amortizado sin usar las técnicas del enfoque de análisis amortizado.

El desafío inicial para el trabajo de esta tesis era mejorar COSTA [7], un analizador de coste basado en el enfoque clásico, para que infiriese UBFs tan precisas como las que se infieren con el enfoque amortizado, pero al mismo tiempo mantuviese sus propiedades de escalabilidad y aplicabilidad.

[4] Esto es, aprender qué problemas se deben considerar y resolver para construir.

[5] El proyecto SPEED solo considera un modelo de coste, el número de iteraciones de bucles, por lo que a sus análisis les llaman *análisis de cotas* o *cotas de alcanzabilidad* [140].

4 Contribuciones y Guión

El Capítulo 9 se describen COSTA y PUBS. COSTA es un analizador de coste y terminación para programas en código byte de JAVA, y PUBS es un resolutor de CRSs. Las contribuciones de esta tesis consisten en mejoras y extensiones de las diferentes fases de COSTA y PUBS, pero también pueden aplicarse a cualquier otro analizador con una arquitectura similar. En los Capítulos 11-13 se describen con más detalle las contribuciones y su relación con las publicaciones que las avalan. Cada uno de los capítulos incluye, además, una discusión del trabajo relacionado, así como del contenido que figura en cada publicación. Por último, el Capítulo 14 concluye y discute el posible trabajo futuro. A continuación describimos brevemente las contribuciones:

Una Transformación Asintótica. Las funciones de cota pueden ser expresiones grande, intrincada e ilegible. Las cotas asintóticas son más pequeñas y simples, y describen de manera sucinta cómo escala el coste en relación con el tamaño de la entrada del programa. Sin embargo, COSTA no infiere cotas asintóticas. Nuestra primera contribución es una transformación automática de las funciones de cota a forma asintótica [2], y un resolutor de CRSs escalable que se basa en esta transformación. Esta contribución se presentó en el artículo

> ELVIRA ALBERT, DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, SAMIR GENAIM, AND GERMAN PUEBLA. Asymptotic Resource Usage Bounds. In Zhenjiang Hu, editor of the Proceedings of *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 5904 of *Lecture Notes in Computer Science*, pages 294–310. Springer, December 2009

En esta disertación, la contribución se discute en el Capítulo 10, y el artículo figura a partir de la Página 195.

Un Análisis de Valores para Operaciones No Lineales. Para inferir cotas precisas, COSTA [7, 1] usa relaciones de valor, como conjunciones de restricciones lineales, que aproximan el valor que las variables del programa pueden tomar durante la ejecución. En nuestra segunda contribución estudiamos las limitaciones de esta aproximación al modelar operaciones aritméticas no lineales, y desarrollamos una técnica escalable que supera estas limitaciones. Esta contribución se presentó en el siguiente artículo:

> DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Handling Non-linear Operations in the Value Analysis of COSTA. *Electronic Notes in Theoretical Computer Science*, 279(1):3–17, 2011. Proceedings of Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)

En esta memoria, la contribución se describe en el Capítulo 11, y el artículo figura a partir de la Página 217.

Un enfoque para tratar la Codependencia entre Salida y Coste. La mayoría de analizadores de coste solo abstraen el coste de un programa como una función de su entrada. Sin embargo, en algunos programas aparece otra codependencia entre la salida y el coste de un procedimiento, y cualquier análisis de coste que ignore dicha codependencia obtiene, inevitablemente, una UBF que es asintóticamente imprecisa. Nuestra tercera contribución da un ejemplo detallado de esta codependencia, y explica cómo afecta a la precisión del análisis de COSTA. Entonces presentamos un enfoque de análisis de coste que mantiene esta codependencia que, usando técnicas de análisis lógico de programas [34], obtiene una UBF más precisa. También presenta técnicas para manejar esta imprecisión, basadas en el uso de resolutores de eliminación de cuantificadores y satisfactibilidad respecto a teorías. Además, también se describe una relación entre esta codependencia y el análisis amortizado de coste. Esta contribución se presentó en este artículo:

> DIEGO ESTEBAN ALONSO-BLAS AND SAMIR GENAIM. On the Limits of the Classical Approach to Cost Analysis. In Antoine Miné and David Schmidt, editors of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 7460 of *Lecture Notes in Computer Science*, pages 405–421. Springer, September 2012

En esta memoria, la contribución se describe en el Capítulo 12, y el artículo figura a partir de la Página 233.

Un método preciso y útil para resolver CRSs. Por último, presentamos un método para resolver CRSs en UBFs, que para muchos CRSs obtiene una UBF es asintóticamente precisa. Este método se basa en las técnicas escalables de eliminación de cuantificadores para fórmulas de aritméticas lineales, lo que hace que este método sea útil y escalable. Esta contribución se presentó en este artículo:

> DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Precise Cost Analysis via Local Reasoning. In Dang Van Hung and Mizuhito Ogawa, editors of the Proceedings of *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 8172 of *Lecture Notes in Computer Science*, pages 319–333. Springer, October 2013

Esta contribución se describe en el Capítulo 13, y el artículo figura a partir de la Página 253.

9 | El Sistema COSTA

Esta tesis presenta técnicas para calcular automáticamente cotas asintóticas precisas en forma cerrada para programas abstractos no deterministas. Estas técnicas se desarrollaron en COSTA. En este capítulo se describe la arquitectura y el funcionamiento interno de COSTA, en la medida en que es necesario para explicar nuestras contribuciones.

1 La Arquitectura de COSTA

COSTA [7, 1] es un analizador automático para Código Byte de JAVA (JAVA *Byte-Code*, JBC), un lenguaje orientado a objetos de bajo nivel [93]. COSTA toma como entrada un programa en JBC, y puede demostrar automáticamente su terminación o calcular una cota sobre su coste respecto a un cierto modelo de coste. Dicha cota es o bien una UBF para el coste en el caso peor, o bien una LBF para el coste en el caso mejor. Dichas cotas son funciones numéricas sobre el tamaño de la entrada del programa, expresadas en forma cerrada, esto es, deterministas y no recursivas.

La Figura 9.1 muestra la arquitectura de COSTA. Los rectángulos con esquinas redondeadas representan los principales componentes, como transformaciones o análisis, y los rectángulos representan formatos de información intermedia que pasa entre estos componentes. Las elipses señalan las publicaciones que soportan esta disertación, y los componentes de COSTA con que se relacionan. En el resto de este capítulo se describe el funcionamiento interno de COSTA, aplicando cada uno de los siguientes componentes a un simple programa JAVA que se introduce en el Ejemplo 9.1:

- La **Representación Basada en Reglas** (*Rule-Based Representation* o RBR) se obtiene por una **decompilación declarativa** del Grafo de Control de Flujo (*Control Flow Graph* o CFG) del programa JBC. Un programa en la RBR es un conjunto de reglas que definen varios procedimientos, donde cada regla corresponde a un bloque básico del programa y cada procedimiento a un método, bucle o bifurcación. La RBR proporciona una representación uniforme del flujo de control del programa, en la forma de reglas con guardas y llamadas interprocedurales, donde todos los bucles se representan con llamadas recursivas. En la Sección 9.2 se describe la RBR y el resultado de la decompilación de nuestro ejemplo.
- El programa en **Reglas Abstractas de Coste** (*Abstract Cost Rules* o ACR) se obtiene desde la RBR mediante una compilación abstracta: la **abstracción**

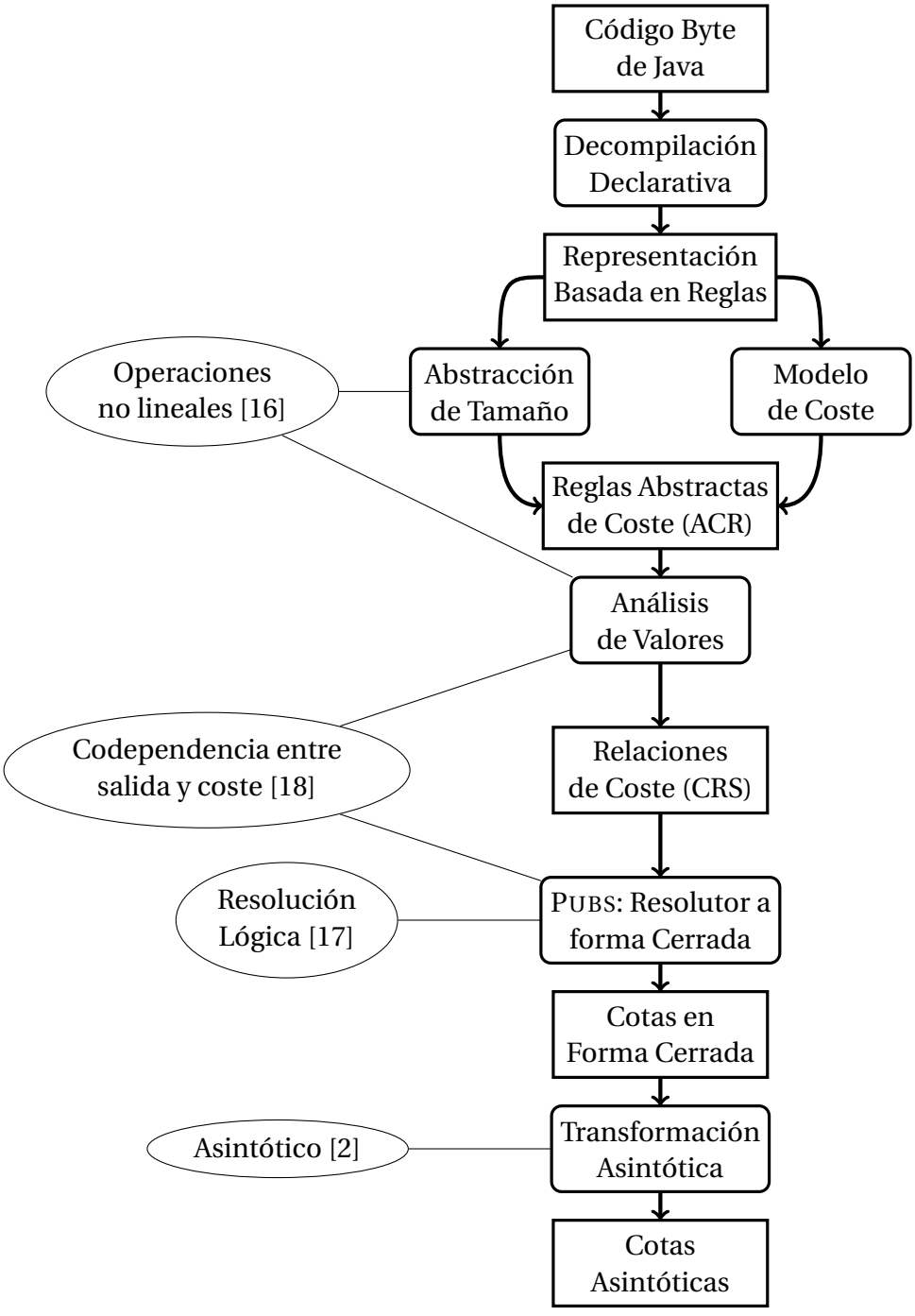


Figura 9.1: La arquitectura de COSTA: análisis, transformaciones, formatos intermedios, y contribuciones de esta tesis.

de tamaños reemplaza cada variable por una variable de tamaño y cada operación por una restricción entre dichas variables de tamaño; y el **modelo de coste** reemplaza cada instrucción de la RBR por una **anotación de coste** que representa el coste de la instrucción. En la Sección 9.3 se describe el lenguaje de los programas ACR, y la compilación abstracta.

- El **Sistema de Relaciones de Coste** (*Cost Relation System* o CRS) se obtiene quitando de la ACR las variables de salida de todas las llamadas interprocedurales, y reemplazándolas con una post condición obtenida por un **análisis de valor**. El CRS solo representa para cada procedimiento la relación entre sus entradas y su coste. En la Sección 9.4 se describe el lenguaje de CRS, y la transformación de la ACR en el CRS.
- El último paso es resolver el CRS en UBFs o LBFs, para lo cual COSTA usa el subsistema PUBS *(Practical Upper Bound Solver)* [5], el cual toma como entrada un CRS y calcula las UBFs o LBFs respectivos. En la Sección 9.5 se describe el proceso de resolución de PUBS.

Aunque COSTA se diseñó para analizar programas JBC, es fácil adaptarlo a otros lenguajes.

Ejemplo 9.1. La Figura 9.2 muestra el código JAVA para el ejemplo guía de este capítulo, una implementación del algoritmo de ordenación por inserción. El método `inSort` ordena los elementos del array entre las posiciones 0 a `last`-1, usando el método auxiliar `insert`. Este método se ha compilado a JBC y analizado con COSTA para obtener cotas en el número de accesos de lectura o escritura a posiciones en el array `data`. Se asume que los accesos a `data[j` -1] en las Líneas 10 y 11 se optimizan a uno, usando una variable auxiliar (esto se ve en la RBR). Para `insert(data,i)`, COSTA calcula la UBF $2 * \mathtt{nat}(i-1)+3$, que se puede explicar así: los accesos en las Líneas 8 y 14 cuestan 2; el bucle `while` contribuye $2 * \mathtt{nat}(i-1)$, que es el producto de 2, el número de accesos en el cuerpo y condición del bucle, por el número de iteraciones $\mathtt{nat}(i-1)$; por último, el acceso en la guarda del bucle, en el caso en que ésta se evalúe a *false*, cuesta 1. En esta UBF se usa el operador $\mathtt{nat}(x)$, que significa máx$(x,0)$. Para `inSort(data,last)`, COSTA calcula la UBF $\mathtt{nat}(\mathit{last}-1) * (3 * \mathtt{nat}(\mathit{last}-1)+3)$. El factor $\mathtt{nat}(\mathit{last}-1)$ acota el número de iteraciones del bucle `for`. El segundo factor da el coste máximo de cada iteración, que es como la UBF de `insert`, salvo que la variable i se reemplaza por $\mathit{last}-1$, esto es, por el máximo valor que el contador del bucle `i` puede tomar. ■

A continuación se describen los pasos que COSTA sigue para inferir estas UBFs. Para abreviar la descripción, los resultados intermedios se han simplificado aunque sin que ello afecte a la corrección de las UBFs. Además, se omite cualquier detalle sobre COSTA que no sea necesario para presentar nuestras contribuciones. Los detalles de COSTA se pueden encontrar en [7, 1].

```
1   // Precondition: 0 <= last <= data.length
2   void inSort(double[] data, int last) {
3       for (int i=1; i<last; i++)
4           insert(data,i);
5   }
6
7   void insert( double[] data, int i){
8       double x = data[i];
9       int j=i;
10      while (j > 0 && data[j - 1] < x) {
11          data[j] = data[j - 1];
12          j--;
13      }
14      data[j] = x;
15  }
```

Figura 9.2: Código JAVA del método `inSort`.

2 Representación Basada en Reglas

COSTA empieza por construir el CFG del programa JBC. El CFG para el (JBC de) los métodos `inSort` e `insert` se muestra en la Figura 9.3. Por brevedad, se omite el comportamiento provocado por el lanzamiento de excepciones. En el CFG, cada rectángulo representa un bloque básico, esto es una secuencia de instrucciones JBC sin bifurcación. Un diamante representa una bifurcación. Un círculo doble representa un final de método o la salida de un bucle. Para abreviar, solo se muestra el nombre de cada procedimiento y las guardas de cada rama.

El CFG se divide en cinco subgrafos o **procedimientos**: uno para el método `inSort`, uno para el bucle `for`, uno para el método `insert`, y dos para el bucle `while` (Líneas 10 a 13). El bucle `while` tiene dos procedimientos porque la condición `data[i−1] < x` (Línea 10) solo se evalúa si `i > 0` es *true*. Cada procedimiento tiene uno o más caminos de control. Un camino de control representa un salto a un bloque básico junto con una **guarda** que especifica en qué casos se toma ese salto. Los procedimientos para los métodos `inSort` e `insert` tienen cada uno un camino de control con la guarda *true*. Los procedimientos que corresponden a bucles tienen dos caminos de control: uno para salir del bucle y uno para ejecutar una iteración. Por ejemplo, en el procedimiento del bucle `for`, el camino de salida lleva la guarda ¬(`i < last`), y el otro lleva la guarda `i < last`.

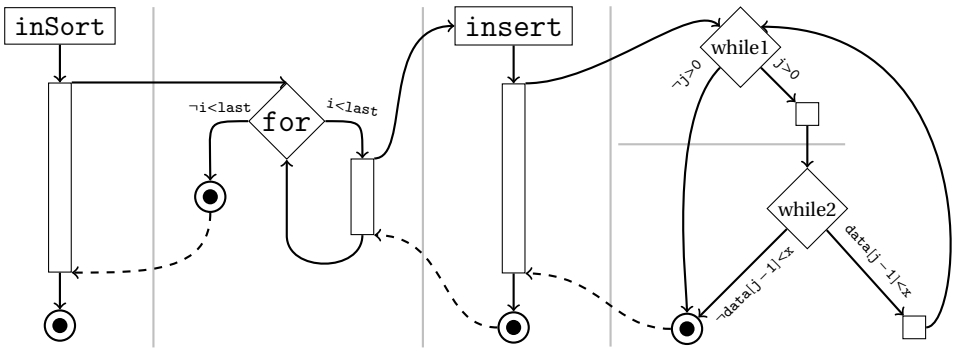


Figura 9.3: Grafo de Control de Flujo para el método `inSort`.

Al generar el CFG, COSTA usa un análisis de punteros [125] para resolver las llamadas virtuales a un método. Además, siempre que es posible, COSTA extrae y aisla cada bucle como si fuera un método. Así, el camino de control de salida del bucle está vacío, en vez de tener las instrucciones ejecutadas tras el bucle. La extracción de bucles [136] es una técnica que COSTA usa para permitir un análisis composicional [7].

Traducción a la RBR

El siguiente paso de COSTA es transformar el CFG en un programa RBR, que consiste en reglas de la forma

$$p(\bar{x}, \bar{y}) \leftarrow \{g\}, b_1, \ldots, b_n.$$

donde p es un nombre de procedimiento, $\bar{x}$ es una lista de variables de entrada, $\bar{y}$ es una lista de variables de salida, g es una **guarda**, y cada b_i es una instrucción RBR.

La guarda es una fórmula lógica definida sobre las variables de entrada, la cual se omite si es *true*. Una instrucción RBR es o bien una llamada $q(\bar{w}, \bar{z})$ al procedimiento q; o una instrucción sin bifurcación que corresponde a una instrucción JBC. Nos restringimos a las instrucciones de asignación que tienen la forma $\tau := e$, donde e es una expresión aritmética sobre variables o accesos a array, y τ es o bien una variable x o bien una posición de array $x[i]$. Nótese que la RBR de COSTA incluye más instrucciones, como son los accesos a campos, o la creación de objetos y arrays. Sin embargo se omiten dichas instrucciones ya que no son necesarias para explicar las contribuciones de la tesis.

Para traducir el CFG a la RBR, cada procedimiento en el CFG se transforma en un procedimiento en la RBR, y cada camino de control en el CFG se transforma en una regla en la RBR, cuya guarda es la misma que la del camino. Los

$inSort(\langle data, last\rangle, \langle\rangle) \leftarrow$
$\quad i := 1,$
$\quad for(\langle data, last, i\rangle, \langle\rangle).$

$for(\langle data, last, i\rangle, \langle\rangle) \leftarrow$
$\quad \{\neg i < last\}.$
$for(\langle data, last, i\rangle, \langle\rangle) \leftarrow$
$\quad \{i < last\},$
$\quad insert(\langle data, i\rangle, \langle\rangle),$
$\quad i := i + 1,$
$\quad for(\langle data, last, i\rangle, \langle\rangle).$

$insert(\langle data, i\rangle, \langle\rangle) \leftarrow$
$\quad x := data[i],$
$\quad j := i,$
$\quad while_a(\langle data, x, j\rangle, \langle j\rangle),$
$\quad data[j] := x.$

$while_a(\langle data, x, j\rangle, \langle j\rangle) \leftarrow$
$\quad \{\neg j > 0\}.$
$while_a(\langle data, x, j\rangle, \langle j\rangle) \leftarrow$
$\quad \{j > 0\},$
$\quad jj := j - 1,$
$\quad y := data[jj],$
$\quad while_b(\langle data, x, j, y\rangle, \langle j\rangle).$

$while_b(\langle data, x, j, y\rangle, \langle j\rangle) \leftarrow$
$\quad \{\neg y < x\}.$
$while_b(\langle data, x, j, y\rangle, \langle j\rangle) \leftarrow$
$\quad \{y < x\},$
$\quad data[j] := y,$
$\quad j := j - 1,$
$\quad while_a(\langle data, x, j\rangle, \langle j\rangle).$

Figura 9.4: Representación Basada en Reglas para el método `inSort`.

saltos entre bloques del CFG se transforman en llamadas a los correspondientes procedimientos. Cada instrucción del programa JBC se transforma en una instrucción en la RBR. Las variables en la RBR tienen como ámbito la regla en la que aparecen, y por lo general corresponden o bien a variables del programa JAVA o a posiciones en la pila de operandos de la JVM.

Ejemplo 9.2. La Figura 9.4 muestra la RBR para nuestro ejemplo guía. Contiene ocho reglas que definen cinco procedimientos, que corresponden a los procedimientos y caminos de control en el CFG de la Figura 9.3.

Los procedimientos *inSort* y *insert* corresponden a los métodos JAVA de la Figura 9.2, y de hecho tienen las mismas entradas y salidas. La regla de *inSort* inicializa la variable i del bucle `for` y llama al procedimiento *for*. La regla de *insert* contiene una instrucción que se corresponde con el acceso al array de la Línea 8, otra para inicializar la variable j del bucle `while`, una llamada al procedimiento $while_a$, y una que corresponde a la escritura en el array de la Línea 14.

Los procedimientos *for*, $while_a$ y $while_b$ corresponden a los bucles `for` y `while` del programa JAVA. Cada uno tiene dos reglas: una para salir del bucle y otra para ejecutar una iteración. Cada iteración del un bucle se realiza usando una llamada recursiva en la RBR. Nótese que los dos accesos a `data[j - 1]` en el bucle `while` se optimizan a un solo acceso en la RBR, guardándose el valor en la variable auxiliar y (en la segunda regla de $while_a$). ■

Semántica de la RBR

En [10] se detalla una semántica de trazas para los programas RBR, cuyos detalles se omiten porque no son importantes para explicar nuestras contribuciones. De manera informal, si se asume que el modelo de coste asigna a cada instrucción RBR algún coste (usando un modelo de coste), el coste de una traza t de la RBR, denotado como $rbrcost(t)$, se define como la suma del coste de todas las instrucciones ejecutadas a lo largo de t. Una función p^+ (o p^-) es una UBF (o LBF) para un procedimiento p, si y solo si para cualquier traza (parcial) t obtenida al ejecutar p sobre una entrada inicial de tamaño $\bar{v}$, se cumple que $p^+(\bar{v}) \geq rbrcost(t)$ (o bien $p^-(\bar{v}) \leq rbrcost(t)$).

La decompilación del JBC a la RBR es correcta ya que hay una correspondencia biunívoca entre las trazas de la RBR y las ejecuciones del JBC. Esto significa que el coste de la RBR es también el coste del programa JBC original.

3 Reglas Abstractas de Coste

El siguiente paso de COSTA es la compilación abstracta del programa RBR a un programa ACR, compuesto por reglas de la siguiente forma:

$$p(\bar{x}^\alpha, \bar{y}^\alpha) \leftarrow a_0, \ldots, a_n.$$

donde p es un nombre de procedimiento; $\bar{x}^\alpha$ e $\bar{y}^\alpha$ son las listas de sus variables (abstractas) de entrada y salida, respectivamente; y cada a_i es una instrucción de la ACR, que puede ser o bien una llamada a un procedimiento $q(\bar{w}^\alpha, \bar{z}^\alpha)$, una restricción (igualdad o desigualdad no estricta) lineal sobre las variables de la regla, o una anotación de coste $\mathsf{acquire}(e)$ que indica que, en ese punto del programa, se suma e al coste de ejecución.

Las variables abstractas en una regla de la ACR son variables enteras que representan el tamaño de las correspondientes variables de la RBR. COSTA usa varias medidas de tamaño que dependen del tipo JBC de la variable RBR: el tamaño de una variable `int` es su valor, el de una variable de tipo array es su longitud, el de una variable de tipo puntero es su longitud de camino [123], y las variables de tipo básico no entero se abstraen como variables "libres" que representan cualquier valor. Si el coste depende de dichas variables, entonces COSTA no puede inferir una UBF para el programa ACR.

La compilación abstracta transforma cada regla "$p(\bar{x}, \bar{y}) \leftarrow \{g\}, b_1, \ldots, b_m$" de la RBR a una regla "$p(\bar{x}^\alpha, \bar{y}^\alpha) \leftarrow a_0, \ldots, a_n$" en la ACR, donde las variables abstractas $\bar{x}^\alpha$ e $\bar{y}^\alpha$ representan los tamaños de $\bar{x}$ e $\bar{y}$; la guarda g se compila a una restricción lineal a_0 sobre las variables abstractas de entrada $\bar{x}^\alpha$, que sobre aproxima su semántica; y cada b_i se compila a uno o más a_j consecutivos, de este modo:

$inSort(\langle l_0\rangle,\langle\rangle) \leftarrow$
  $i_2 = 1$,
  $for(\langle l_0, i_2\rangle,\langle\rangle)$.

$for(\langle l_0, i_0\rangle,\langle\rangle) \leftarrow$
  $i_0 \geq l_0$.
$for(\langle l_0, i_0\rangle,\langle\rangle) \leftarrow$
  $i_0 + 1 \leq l_0$,
  $insert(\langle i_0\rangle,\langle\rangle)$,
  $i_2 = i_0 + 1$,
  $for(\langle l_0, i_2\rangle,\langle\rangle)$.

$insert(\langle i_0\rangle,\langle\rangle) \leftarrow$
  acquire(1),
  $while_a(\langle i_0\rangle,\langle\rangle)$,
  acquire(1).

$while_a(\langle j_0\rangle,\langle\rangle) \leftarrow$
  $j_0 \leq 0$.
$while_a(\langle j_0\rangle,\langle\rangle) \leftarrow$
  $j_0 \geq 1$,
  acquire(1),
  $while_b(\langle j_0\rangle,\langle\rangle)$.
$while_b(\langle j_0\rangle,\langle\rangle) \leftarrow true$.
$while_b(\langle j_0\rangle,\langle\rangle) \leftarrow$
  acquire(1),
  $j_2 = j_0 - 1$,
  $while_a(\langle j_2\rangle,\langle\rangle)$.

Figura 9.5: Programa ACR para el método `inSort`.

- **Abstracción de tamaño.** Si b_i es una llamada $q(\bar{w},\bar{z})$ entonces se compila a $q(\bar{w}^\alpha,\bar{z}^\alpha)$; si es una asignación de la forma $x := e$, donde e es una expresión lineal, entonces se compila a $x^\alpha = e^\alpha$, donde e^α se obtiene a partir de e reemplazando sus variables por sus correspondientes abstracciones; si e no es una expresión lineal entonces se compila a una restricción *true*, la cual no afecta a la ejecución. La abstracción de tamaño usa una transformación conocida como *Static Single Assignment* (SSA) [21, §19] que modela el efecto de las asignaciones con restricciones lineales. Lógicamente, como la RBR de COSTA soporta otras instrucciones, tales como el acceso a campos, la abstracción de tamaño que se usa en COSTA es más sofisticada de lo que se ha descrito. Sin embargo estos detalles no son relevantes en el contexto de la tesis, ya que vamos a partir de un programa ACR dado, independientemente de donde provenga.
- **Anotaciones de Coste.** El **modelo de coste** se aplica a b_i para generar una anotación de coste de la forma acquire(e), que describe el coste de la instrucción RBR. En la ACR se omiten estas anotaciones cuando $e = 0$. COSTA tiene varios modelos de coste, tales como el número de instrucciones, de llamadas a un método, o uso de memoria.

La compilación abstracta termina eliminando de la ACR las variables que no afectan al coste [8], para así obtener un programa ACR más conciso y fácil de analizar.

Ejemplo 9.3. La Figura 9.5 muestra el programa ACR obtenido por la compilación abstracta del programa RBR en la Figura 9.4. La abstracción de tamaño proyecta las variables *last*, i, y j de la RBR a las variables l, i y j de la ACR, donde aparecen con subíndices añadidos por la SSA. Las variables x, y, *data* de la RBR y la variable de salida de los procedimientos $while_a$ y $while_b$ se omiten porque

no afectan al coste.

La relación entre cada regla en la RBR y su correspondiente regla en la ACR es evidente. La asignación $i := i+1$ en la segunda regla del procedimiento *for* de la RBR se compila a la restricción $i_2 = i_0 + 1$, donde i_0 e i_2 son los valores de la variable i antes y después de la ejecución de la instrucción. Cada acceso a un array se cuenta con la anotación de coste acquire(1). ■

Semántica de la ACR

Resumamos brevemente los detalles clave de la semántica de los programas ACR [18, 10]. Un estado s es de la forma $\langle \psi, \bar{a} \rangle$, donde $\bar{a}$ es la secuencia de instrucciones ACR a ejecutar, y ψ es una restricción sobre las variables en $\bar{a}$. En esencia, ψ guarda las restricciones encontradas durante la ejecución. Una ejecución comienza con un estado inicial $\langle \bar{x} = \bar{v}, p(\bar{x}, \bar{y}) \rangle$, donde $\bar{v}$ es una secuencia de enteros, y procede según las siguientes reglas:

$$\frac{q(\bar{x},\bar{y}) \leftarrow \bar{a}' \in P}{\langle \psi, q(\bar{x},\bar{y}) \cdot \bar{a} \rangle \overset{0}{\rightsquigarrow} \langle \psi, \bar{a}' \cdot \bar{a} \rangle} \qquad \frac{\psi \wedge \varphi \not\models \mathit{false}}{\langle \psi, \varphi \cdot \bar{a} \rangle \overset{0}{\rightsquigarrow} \langle \psi \wedge \varphi, \bar{a} \rangle} \qquad \frac{\mathit{eval}(e,\psi) = v \geq 0}{\langle \psi, \mathsf{acquire}(e) \cdot \bar{a} \rangle \overset{v}{\rightsquigarrow} \langle \psi, \bar{a} \rangle}$$

Éstas definen una relación de transición $s_1 \overset{v}{\rightsquigarrow} s_2$, que indica que hay una transición de s_1 a s_2 que cuesta v unidades. La regla de la izquierda se aplica en presencia de llamadas a procedimientos, para lo que escoge una regla en el programa P del procedimiento llamado, y se añaden sus instrucciones $\bar{a}'$ a la secuencia de instrucciones pendientes. Las variables en $\bar{a}'$ (salvo $\bar{x} \cup \bar{y}$) se renombran para que sean distintas de las variables en $\bar{a}$ y ψ. La regla del centro es aplicable a las restricciones, añadiéndolas a ψ siempre que la conjunción sea satisfactible. La regla de la derecha maneja las anotaciones de coste, evaluando e a un valor $v \geq 0$, que representa el coste de la transición.

La ejecución finaliza cuando ninguna regla es aplicable. Esto sucede si se alcanza, o bien un estado final $\langle \psi', \epsilon \rangle$, donde ϵ representa la secuencia vacía, o bien un estado de bloqueo $\langle \psi', \varphi \cdot \bar{a} \rangle$, donde $\varphi \wedge \psi' \models \mathit{false}$. Una traza t es una secuencia de estados con una transición válida entre cada dos estados consecutivos. Una traza es completa si acaba en un estado final o es infinita. No se consideran trazas que acaban en un estado de bloqueo ya que éstas no se corresponden con ninguna traza de la RBR. El coste de una traza t, denotado como $\mathit{acrcost}(t)$, es la suma de los costes de sus transiciones. Una función p^+ (o p^-) es una UBF (o LBF) para un procedimiento p si, para cualquier entrada $\bar{v}$ y para cualquier traza completa t que empiece en el estado $\langle \bar{x} = \bar{v}, p(\bar{x}, \bar{y}) \rangle$, se cumple que $p^+(\bar{v}) \geq \mathit{acrcost}(t)$ (o bien $p^-(\bar{v}) \leq \mathit{acrcost}(t)$).

La transformación de la RBR a la ACR es correcta ya que para cualquier traza de la RBR hay al menos una traza de la ACR con el mismo coste. Por lo tanto una UBF (o LBF) del programa ACR lo es también para el programa RBR.

4 Sistemas de Relaciones de Coste

La siguiente fase en COSTA transforma el programa ACR en un CRS, compuesto por ecuaciones de la forma

$$C(\bar{x}) = e + D_1(\bar{y}_1) + \cdots + D_r(\bar{y}_r)\ , \varphi$$

donde $C, D_1, \ldots, D_r$ son símbolos de relación (como los nombres de procedimiento); $\bar{x}, \bar{y}_1, \ldots, \bar{y}_r$ son variables; φ es una conjunción (escrita como conjunto) de restricciones lineales sobre esas variables; y e es una **expresión de coste** formada a partir de la siguiente gramática:

Expresión de Coste	$\mathsf{exp} ::= \mathsf{bexp} \mid \mathsf{oexp}$
Expresión Básica	$\mathsf{bexp} ::= q \mid \mathsf{nat}(l) \mid \log_m(\mathsf{nat}(l)+1) \mid m^{\mathsf{nat}(l)}-1$
Expresión Compuesta	$\mathsf{oexp} ::= \mathsf{exp}+\mathsf{exp} \mid \mathsf{exp}*\mathsf{exp} \mid \mathsf{máx}(\mathsf{exp}_1, \ldots, \mathsf{exp}_n)$

donde $q \in \mathbb{Q}^+$, $m > 1 \in \mathbb{Z}^+$, l es una expresión lineal sobre las variables de la ecuación, y $\mathsf{nat}(l) = \mathsf{máx}(l, 0)$. Intuitivamente, esta ecuación indica que el coste de C, para la entrada $\bar{x}$, es e más el coste de cada D_i para la entrada $\bar{y}_i$. La restricción φ especifica para qué valores de $\bar{x}$ es aplicable la ecuación y qué relación hay entre las variables. Dado que un CRS se genera a partir de un programa ACR, se puede pensar en $C, D_1, \ldots, D_r$ como si fueran procedimientos no deterministas y decir que C llama a $D_1, \ldots, D_r$.

Las ecuaciones con el mismo lado izquierdo $C(\bar{x})$ definen una relación $C(\bar{x})$, a la que llamamos **relación de coste** (*Cost Relation* o CR). Así, un CRS contiene varias CRs. Si una CR solo usa un símbolo de relación, $D_i = C$ para $1 \le i \le r$, se dice que es una CR **aislada**.

La transformación de la ACR en el CRS proyecta cada procedimiento $p(\bar{x}, \bar{y})$ de la ACR a una CR $p(\bar{x})$ en el CRS, la cual tiene las mismas entradas $\bar{x}$, pero sin las salidas $\bar{y}$. Esto se hace transformando cada regla ACR "$p(\bar{x}, \bar{y}) \leftarrow a_1, \ldots, a_n$" en una ecuación "$p(\bar{x}) = e + q_1(\bar{w}_1) + \cdots + q_n(\bar{w}_n), \varphi$", donde:

1. la expresión de coste e es la suma de todas las expresiones e_i que aparecen en las anotaciones $\mathsf{acquire}(e)$ de la regla ACR;
2. cada llamada $q_j(\bar{w}_j, \bar{z}_j)$ en la regla ACR rule se transforma en una llamada $q_j(\bar{w}_j)$ en la ecuación, con las mismas entradas pero sin las salidas; y
3. φ contiene todas las restricciones lineales en la regla ACR.

En el segundo paso, cuando $q_j(\bar{w}_j, \bar{z}_j)$ tiene al menos una variable de salida, al quitarla se pierde información que puede ser crucial para inferir una UBF. Para reducir el efecto de esta pérdida, COSTA usa un **análisis de valores** interpro-

$$
\begin{array}{rcll}
inSort(l_0) & = & for(l_0, i_2) & \{i_2 = 1\} \\
for(l_0, i_0) & = & 0 & \{i_0 \geq l_0\} \\
for(l_0, i_0) & = & insert(i_0) + for(l_0, i_2) & \{i_0 + 1 \leq l_0, i_2 = i_0 + 1\} \\
insert(i_0) & = & 2 + while_a(i_0) & \{\} \\
while_a(j_0) & = & 0 & \{j_0 \leq 0\} \\
while_a(j_0) & = & 1 + while_b(j_0) & \{j_0 \geq 1\} \\
while_b(j_0) & = & 0 & \{\} \\
while_b(j_0) & = & 1 + while_a(j_2) & \{j_2 = j_0 - 1\}
\end{array}
$$

Figura 9.6: Sistema de Relaciones de Coste (CRS) para el método `inSort`.

cedural [1] que calcula para cada procedimiento de la ACR una post condición, y ésta se añade a φ para compensar la eliminación de las variables de salida. Dicha post condición es una conjunción de restricciones lineales.

Ejemplo 9.4. La Figura 9.6 muestra el CRS generado desde el programa ACR de la Figura 9.5. El CRS define cinco CRs en ocho ecuaciones. En cada ecuación, la expresión de coste coincide con las anotaciones de coste de la regla correspondiente, siendo 2 para *insert*, 1 para las reglas recursivas de $while_a$ y $while_b$, y 0 en el resto; las llamadas son como las de la ACR; y las restricciones son las que aparecen en la regla. Para este ejemplo no hace falta usar el análisis de valores porque ningún procedimiento de la ACR tiene variables de salida. ■

Semántica del CRS

La semántica del CRS se basa en la noción de árbol de evaluación [5]. Denotemos un árbol (quizá infinito) como $node(q, \langle T_1, \ldots, T_k \rangle)$, donde $q \in \mathbb{Q}^+$ es el valor de la raíz y $T_1, \ldots, T_k$ son subárboles. Dado una *CR* C y unos valores de entrada $\bar{v}$, se dice que $node(v_e, \langle T_1, \ldots, T_k \rangle)$ es un árbol de evaluación para $C(\bar{v})$ si y solo si existe una ecuación "$C(\bar{x}) = e + \sum_{j=1}^{k} D_j(\bar{y}_j), \varphi$" y una asignación σ para las variables de esa ecuación tal que:

1. $\sigma(\bar{x}) = \bar{v}$ y σ satisface φ;
2. e se evalúa a v_e en la asignación σ; y
3. cada T_i es un árbol de evaluación para $C(\sigma(\bar{y}_i))$.

Intuitivamente, si se ve C como un procedimiento, un árbol de evaluación puede verse como un árbol de recursión, donde la llamada $C(\bar{v})$ se evalúa como sigue: se coge una ecuación de C y una asignación σ que satisface φ; se evalúa

[1] La noción de análisis de valor está relacionada con el *análisis de tamaño*, el cual infiere relaciones no entre los valores de las variables numéricas sino entre los tamaños de estructuras de datos.

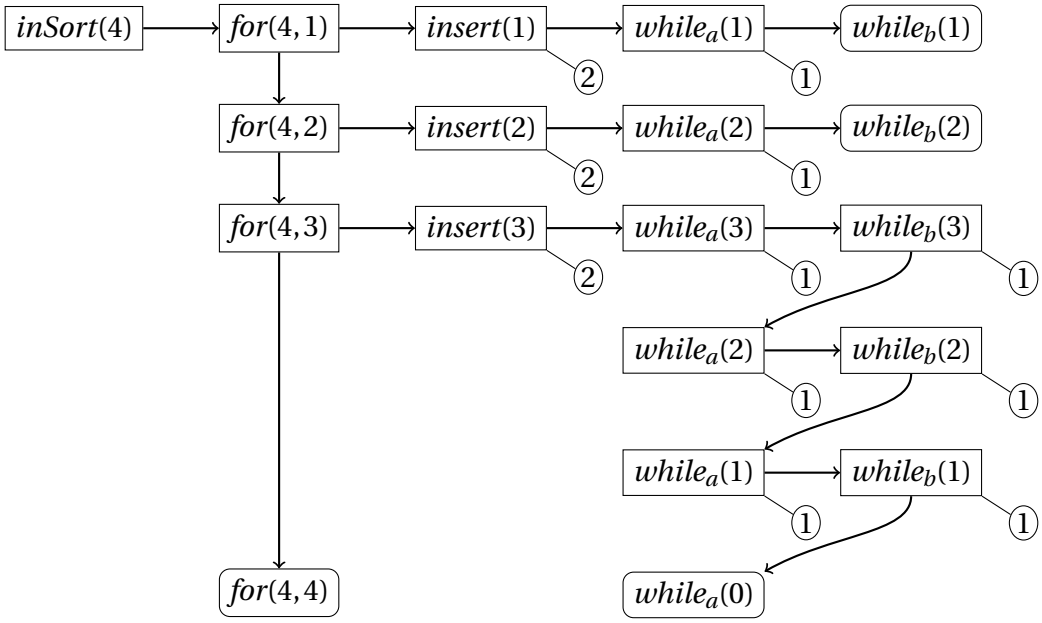


Figura 9.7: Árbol de Evaluación para una llamada a la CR *inSort* de la Figura 9.6

e a v_e, y se construye un árbol para cada $C(\sigma(\bar{y}_i))$. En un árbol de evaluación, las hojas corresponden a aplicaciones de las ecuaciones no recursivas y los nodos internos a aplicaciones de las ecuaciones recursivas. Un árbol de evaluación puede ser infinito. A causa del indeterminismo en los pasos anteriores, $C(\bar{v})$ puede tener varios árboles de evaluación.

Se denota con $\mathit{Trees}(C(\bar{v}))$ el conjunto de todos los árboles de evaluación para $C(\bar{v})$, y el conjunto de todos los posibles costes por $\mathit{Answers}(C(\bar{v})) = \{\mathsf{Sum}(T) \mid T \in \mathit{Trees}(C(\bar{v}))\}$, donde $\mathsf{Sum}(T)$ es la suma de todos los nodos de T. Una función C^+ (o C^-) es una UBF (o LBF) para la CR C si y solo si para cualquier entrada $\bar{v}$ y $c \in \mathit{Answers}(C(\bar{v}))$, se cumple que $C^+(\bar{v}) \geq c$ (o bien $C^-(\bar{v}) \leq c$).

Ejemplo 9.5. La Figura 9.7 muestra un árbol de evaluación para una llamada a $\mathit{inSort}(4)$ en el CRS de la Figura 9.6. Cada rectángulo representa una llamada en el CRS con unos valores de entrada, cada círculo indica el coste (v_e) añadido por la ecuación usada para resolver la llamada; y las aristas indican llamadas a otras CRs. El coste de este árbol de evaluación es 11, que es la suma de todos los círculos. ■

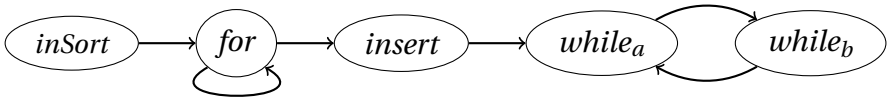
La transformación de la ACR en un CRS es correcta porque cada para cada traza ACR existe un árbol de evaluación con el mismo coste [10]. Por tanto, una UBF o LBF válida para el CRS también lo es para el programa ACR.

5 Cotas Superiores en Forma Cerrada

En el último paso COSTA resuelve el CRS generando una UBF o LBF para cada CR. Para ello, usa el subsistema PUBS [5, 14]. Se describen cómo se calculan las UBFs. El cálculo de las LBFs se comenta brevemente al final del capítulo, ya que no es relevante para la tesis.

El procedimiento de resolución de PUBS está diseñado para utilizar CRSs en los que todas las recursiones son directas, es decir, no existe recursión mutua. Para resolver un CRS que tiene recursión mutua, PUBS primero lo transforma a recursión directa [85, 139]. Para realizar la transformación, se considera cada grupo de CRs mutuamente recursivas, que coinciden con las SCC recursivas en el grafo de llamadas del CRS, y después se despliegan todas las CRs en el grupo en una de ellas.

Ejemplo 9.6. Consideremos el CRS de la Figura 9.6, y su correspondiente grafo de llamadas:



Este grafo tiene dos SCCs recursivas: una para la CR *for*, la cual ya está en recursión directa, y otra para las CRs $while_a$ y $while_b$, las cuales son mutuamente recursivas. Para transformar éstas a recursión directa, se despliegan las ecuaciones de $while_b$ en las de $while_a$. Al hacerlo se obtiene una nueva definición de la CR $while_a$:

$$\begin{array}{ll} while_a(j_0) = 0 & \{j_0 \leq 0\} \\ while_a(j_0) = 1 & \{j_0 \geq 1\} \\ while_a(j_0) = 2 + while_a(j_2) & \{j_0 \geq 1, j_2 = j_0 - 1\} \end{array}$$

La primera ecuación es como la primera ecuación de $while_a$. La segunda y tercera ecuaciones se obtienen de la segunda ecuación del antiguo $while_a$, al desplegar la llamada a $while_b$ usando la primera y segunda ecuaciones de $while_b$, respectivamente. La expresión de coste en la tercera nueva ecuación es la suma de las expresiones de coste de las ecuaciones de $while_a$ y $while_b$, y sus restricciones son las de las ecuaciones recursivas de ambas CRs. ■

Supongamos ahora que el CRS ya está en recursión directa. El procedimiento de PUBS es un proceso iterativo, que resuelve todas las CRs de una en una. En cada iteración se resuelve una CR aislada en una UBF, que luego se substituye en cualquier llamada a esa CR, generando así más CRs aisladas. Este proceso se repite hasta que todas las CRs están resueltas. Nótese que un CRS con recursión directa siempre hay al menos una CR aislada.

Resolviendo una CR aislada

Sea C una CR aislada con n ecuaciones, y (1) sea bf el número máximo de llamadas recursivas en cualquier ecuación de C, que coincide con el máximo factor de ramificación (esto es el número de hijos en cada nodo) los árboles de evaluación de esa CR; (2) sean $e_1, \ldots, e_k$ las expresiones de coste de las ecuaciones no recursivas de C; y (3) sean $e_{k+1}, \ldots, e_n$ las expresiones de las ecuaciones recursivas de C.

El funcionamiento de PUBS para resolver C y generar una UBF es pesimista, es decir se basa en construir un árbol de evaluación cuyo coste es más grande que el coste de cualquier árbol de evaluación de C. Para cualquier árbol de evaluación, el coste de cada nodo interno es una instancia de algún e_i con $k+1 \leq i \leq n$, y el coste de cada hoja es una instancia de e_j con $1 \leq j \leq k$. Ahora supongamos que se dispone de:

- una expresión de coste $h(\bar{x})$ que acota la altura de cualquier árbol de evaluación. Es decir, para cualquier entrada $\bar{v}$ y cualquier árbol de evaluación $T \in \mathit{Trees}(C(\bar{v}))$, $h(\bar{v})$ es mayor que la altura de T; y
- una expresión de coste $\hat{e}_i(\bar{x})$, para cada $1 \leq i \leq n$, que acota el coste aportado por la ecuación i. Esto es, para cualquier entrada $\bar{v}$ y cualquier árbol de evaluación $T \in \mathit{Trees}(C(\bar{v}))$, $\hat{e}_i(\bar{v})$ es mayor que el coste de cualquier nodo en T correspondiente a la aplicación de la ecuación i.

Entonces se construye un árbol (pesimista) de evaluación T para $C(\bar{x})$ de la siguiente forma:

1. T es un árbol completo con factor de ramificación bf y altura $h(\bar{x})$;
2. cada hoja de T tiene como coste $\max(\hat{e}_1(\bar{x}), \ldots, \hat{e}_k(\bar{x}))$; y
3. cada nodo interno de T tiene coste $\max(\hat{e}_{k+1}(\bar{x}), \ldots, \hat{e}_n(\bar{x}))$.

El número de hojas $\mathscr{L}$ en T es $bf^{h(\bar{x})}$, y el número de nodos internos $\mathscr{N}$ es $\frac{bf^{h(\bar{x})}-1}{bf-1}$ si $bf > 1$ y $h(\bar{x})$ en otro caso. El coste de T es:

$$C^+(\bar{x}) = \mathscr{L} * \max(\hat{e}_{k+1}(\bar{x}), \ldots, \hat{e}_n(\bar{x})) + \mathscr{N} * \max(\hat{e}_1(\bar{x}), \ldots, \hat{e}_k(\bar{x}))$$

que es de hecho una UBF correcta para C.

Para inferir automáticamente $h(\bar{x})$, PUBS se basa en el uso de **funciones de rango lineales** [26], que se usan frecuentemente para acotar el número de iteraciones de un bucle. Es fácil observar que la altura de un árbol se corresponde con el número de llamadas recursivas consecutivas en C, que es una forma de bucle. Para inferir automáticamente $\hat{e}_i(\bar{x})$, PUBS usa un procedimiento de **maximización** basado en el cálculo de invariantes de un bucle. Los detalles técnicos sobre el proceso de maximización se pueden encontrar en [5].

Ejemplo 9.7. Veamos ahora los pasos que PUBS sigue para resolver el CRS de la Figura 9.6, una vez éste se ha transformado a recursión directa en el Ejemplo 9.6:

1. Primero se resuelve la CR $while_a$ del Ejemplo 9.6, pues es la única CR aislada. La altura de sus árboles de evaluación está acotada por $h(j_0) = \mathsf{nat}(j_0)$, y los costes de la ecuación recursiva y no recursiva son 1 y 2, respectivamente, por lo que se obtiene como UBF ${while_a}^+(j_0) = 2 * \mathsf{nat}(j_0) + 1$.
2. Al sustituir ${while_a}^+(j_0)$ en la CR no recursiva para *insert*, se obtiene una CR aislada definida por una única ecuación:

$$insert(i_0) = 2 + 2 * \mathsf{nat}(i_0) + 1 \quad \{\}$$

   que se resuelve trivialmente en la UBF $insert^+(i_0) = 2 * \mathsf{nat}(i_0) + 3$.
3. Sustituyendo $insert^+(i_0)$ en la CR *for*, se obtiene la siguiente CR aislada:

$$\begin{array}{ll} for(l_0, i_0) = 0 & \{i_0 \geq l_0\} \\ for(l_0, i_0) = 2 * \mathsf{nat}(i_0) + 3 + for(l_0, i_2) & \{i_0 + 1 \leq l_0, i_2 = i_0 + 1\} \end{array}$$

   La altura de los árboles de evaluación de *for* está acotada por $h(l_0, i_0) = \mathsf{nat}(l_0 - i_0)$. La expresión de coste $2 * \mathsf{nat}(i_0) + 3$ se maximiza a $\hat{e}(l_0, i_0) = 2 * \mathsf{nat}(l_0 - 1) + 3$ porque el valor de i_0 puede ser a lo sumo $l_0 - 1$, donde l_0 se refiere al valor en la llamada inicial a *for*, no al parámetro de la ecuación. Así se obtiene la UBF $for^+(l_0, i_0) = \mathsf{nat}(l_0 - i_0) * (2 * \mathsf{nat}(l_0 - 1) + 3)$.
4. Al sustituir esta UBF en *inSort* se obtiene la siguiente CR aislada:

$$inSort(l_0) = \quad \mathsf{nat}(l_0 - i_2) * (2 * \mathsf{nat}(l_0 - 1) + 3) \quad \{i_2 = 1\}$$

   Como en este caso la ecuación es no recursiva, la altura de cualquier árbol de evaluación es 0. Por otro lado la maximización de $\mathsf{nat}(l_0 - i_2) * (2 * \mathsf{nat}(l_0 - 1) + 3)$ es $\mathsf{nat}(l_0 - 1) * (2 * \mathsf{nat}(l_0 - 1) + 3)$ ya que $i_2 = 1$. Por lo tanto se obtiene finalmente la UBF $inSort^+(l_0) = \mathsf{nat}(l_0) * (2 * \mathsf{nat}(l_0 - 1) + 3)$.

Nótese que estas UBFs son asintóticamente precisas, si se tiene en cuenta que el caso peor de `inSort` es cuadrático en el parámetro `last`. ■

La inferencia de LBFs en PUBS se hace como se describe en [14], aproximando el CRS de entrada con una relación de recurrencia y usando un sistema de álgebra por computador para resolverlo. Para nuestro ejemplo, se infiere la LBF $inSort^-(l_0) = 3 * \mathsf{nat}(l_0 - 1)$, que corresponde al caso en que el array de entrada ya está ordenado y por lo tanto nunca entra en el cuerpo del bucle `while`.

10 | Cotas Asintóticas

En este Capítulo se describe un método para transformar una UBF a una forma reducida asintótico, y cómo usar esta transformación para construir un **analizador de coste asintótico**, que calcula directamente cotas en forma asintótica. Esta contribución se presentó en el artículo:

> ELVIRA ALBERT, DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, SAMIR GENAIM, AND GERMAN PUEBLA. Asymptotic Resource Usage Bounds. In Zhenjiang Hu, editor of the Proceedings of *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 5904 of *Lecture Notes in Computer Science*, pages 294–310. Springer, December 2009

que se puede encontrar en la Página 195.

Descripción

Las notaciones asintóticas se usan para describir sucintamente cómo el coste de ejecución de un programa escala con el tamaño de sus entradas. La observación clave en ellas es que:

> "Aunque a veces se puede determinar el [coste] exacto de un algoritmo [...], tal precisión no compensa el esfuerzo para calcularlo. Para entradas lo bastante grandes, las constantes multiplicativas y los términos de menor orden de la fórmula de coste exacto están dominados por el mero efecto de la entrada". [45, Chapter 3]

La notación asintótica permite ignorar dichas constantes y términos menores, y se centra en describir la proporción, con algún escalar y para entradas a partir de un tamaño, entre el coste de un programa en el caso peor (o en el caso mejor) y una función de los tamaños de la entrada de ese programa. Esta proporción o **equivalencia asintótica** es una condición tan laxa que el caso peor (o el caso mejor) puede ser equivalente asintóticamente a muchas funciones, pero de entre éstas hay una muy simple, como 1, n^2, $n \log n$, o 2^n, a la que se llama la **forma asintótica**. Una cota en forma cerrada sobre el coste de un programa en forma asintótica se llama una **costa asintótica en forma cerrada** [1], esto es, una UBF asintótica (*asymptotic* UBF o AUBF) o una LBF asintótica (o ALBF).

[1] También llamada complejidad asintótica del programa. Para ciertos modelos de coste, se usan los términos complejidad en tiempo o en espacio.

En algunas aplicaciones se necesita una cota con todas sus constantes multiplicativas y términos de menor grado, esto es, una cota no asintótica. Por ejemplo, para repartir tareas entre procesadores [95], para decidir si y a dónde migrar un proceso [55], para no agotar la batería [86], o para evitar un desbordamiento de memoria [13], no se puede usar tal información como que a partir de cierto punto el coste es linear por alguna constante. En general, se necesitan cotas no asintóticas para la verificación y la compilación orientada a la optimización de programas.

En esas aplicaciones normalmente se trabaja con programas ya construidos. Sin embargo, se sabe [91] que no es conveniente esperar hasta después del desarrollo antes de comprobar (automáticamente) si un programa cumple su especificación (en este caso de coste) o no. Esta comprobación debe hacerse con frecuencia, para así detectar los errores de rendimiento lo antes posible. Esto puede hacerse, por ejemplo, ejecutando el análisis y reportando a los programadores las cotas inferidas; o bien éstos pueden escribir en el código una aserción de coste para un método y usar el analizador para verificarla. En ambos casos, usar cotas asintóticas en vez de no asintóticas tiene varias ventajas:

- Al ser más concisas, una cota asintótica es más legible que una no asintótica, lo que mejora la usabilidad para los programadores del análisis.
- Antes de escribir el programa, los programadores están ante todo interesados en su escalabilidad, y pueden tener al respecto una idea lo bastante clara como para escribirla en una aserción de cota asintótica. Así pues, tales aserciones son fáciles de averiguar. En cambio, si se les pide escribir una aserción de cota no asintótica, podrían no saber qué coeficientes y expresiones menores poner en ella, ya que dicha cota refleja detalles sobre la implementación que no pueden saber antes de escribir el programa.
- Mientras se está desarrollando un programa, una cota no asintótica es una información volátil, ya que cualqueir cambio en el programa, el compilador o las bibliotecas puede afectar a las *constantes multiplicativas y términos de orden inferior*. Y dado que esas cosas pasan a menudo durante el desarrollo, si se usaran cotas no asintóticas entonces habría demasiadas notificaciones y demasiado frecuentes. En cambio, las cotas asintóticas son más estables, ya que solo grandes cambios o mejoras pueden moficiar la complejidad asintótica del programa.
- Una vez que se ha escrito un programa, los programadores tienen que mejorar su rendimiento y escalabilidad. Una guía para esta tarea es centrarse en los cuellos de botella del programa, esto es, en las operaciones más caras y frecuentes:

  > "Los programadores pierden un montón de tiempo preocupándose sobre [...] la velocidad de las partes menos importantes de

> sus programas, y buscar la eficiencia en ellas a menudo tiene un efecto negativo si se consideran las tareas de depuración y mantenimiento. *Deberíamos* olvidarnos de las pequeñas eficiencias, digamos, el 97 % del tiempo: la optimización prematura es la raíz del mal. [...] Un buen programador [...] hará bien en mirar con atención el código crítico, pero solo *después* de haberlo encontrado. [87, page 268]"

Un analizador de coste asintótico puede ayudarnos a encontrar el código crítico en el que centrar los esfuerzos de optimización.

- Como las cotas asintóticas son más concisas, un analizador de coste asintótico puede ser más rápido y escalable que uno no asintótico. Tener un analizador más rápido es especialmente relevante para integrarlo en una herramienta interactiva que compila y analiza el código según se escribe.

Así pues, un analizador de coste asintótico puede ser una herramienta sencilla para analizar y mejorar el rendimiento del programa.

En la Sección 10.1, se repasan las definiciones clásicas de las notaciones asintóticas, sus propiedades más relevantes, y se generalizan para las expresiones de coste de PUBS. En la Sección 10.2 se describe la transformación asintótica, que es un procedimiento automático para transformar una expresión de coste en otra asintóticamente equivalente. En la Sección 10.3 se describe la comparación asintótica, que es un procedimiento automático para probar si una expresión de coste es asintóticamente mayor que otra. En la Sección 10.4 se ve cómo usar estos procedimientos para construir un analizador de coste asintótico. En la Sección 10.5 se discute una evaluación experimental de nuestras técnicas. En la Sección 10.6 se repasa el trabajo relacionado, y finalmente en la Sección 10.7 se mencionan algunos detalles que aparecen en el artículo.

1 Notaciones Asintóticas para Expresiones de Coste

En esta sección se recuerdan las definiciones de las notaciones asintóticas para funciones monovariable y sus propiedades más relevantes, y éstas se adaptan a las expresiones de coste de PUBS (Sección 9.4).

Las notaciones asintóticas big Omicron (O) y big Theta (Θ), se definen habitualmente de la siguiente forma: si g es una función monovariable, esto es una función en $\mathbb{N} \mapsto \mathbb{R}^+$, entonces $O(g)$ y $\Theta(g)$ denotan conjuntos de funciones de una variable. En concreto, $f \in O(g)$ si para cualquier entrada n lo suficientemente grande, se cumple que $f(n) \leq c_u g(n)$, para alguna constante $c_u > 0$. Del mismo modo, $f \in \Theta(g)$ si $c_l g(n) \leq f(n) \leq c_u g(n)$ para valores grandes de la entrada n y para algunos coeficientes $c_l, c_u > 0$.

Estas notaciones se suelen utilizar para comparar funciones, y por lo tanto $f \in O(g)$ se puede interpretar como "f es asintóticamente menor que g", o "g es una cota superior asintótica de f"; asimismo se puede interpretar $f \in \Theta(g)$ como que "f es asintóticamente equivalente a g", o "f es una cota superior ajustada de f". Con este enfoque, $f \in O(g)$ y $f \in \Theta(g)$ pueden verse como relaciones de órden [2], que cumplen las siguientes propiedades:

1. El orden asintótico Θ de una función no cambia al multiplicar la función por una constante $d > 0$, es decir, $\Theta(d * f) = \Theta(f)$.
2. El producto de funciones es monótono respecto a la relación de orden definida por O. Formalmente, si $f' \in O(f)$ y $g' \in O(g)$ entonces $f' * g' \in O(f * g)$.
3. El orden de una suma es aquél del término dominante. Formalmente, si $f \in O(g)$ entonces $\Theta(f + g) = \Theta(g)$.

Estas propiedades son esenciales para poder implementar una transformación o comparación asintótica automática, ya que de ellas se deducen algunas características fáciles de implementar. Por ejemplo, la regla referente a la comparación de dos polinomios utilizando su grado (lineal, cuadrático, cúbico), o la que permite comparar dos exponenciales por su base [3]. Debido a esas propiedades, el conjunto Θ asociado a una función tan simple como n o n^2, puede contener funciones más complejas. El siguiente ejemplo islustra este hecho.

Ejemplo 10.1. Consideremos la función $f(n) = 3n^2 + n\log_3 n + 5n$. Podemos omitir los coeficientes multiplicativos 3 y 5 gracias a la primera propiedad. Enconces, puesto que n y $n\log_3 n$ pertenecen a $O(n^2)$, usando la tercera propiedad se obtiene que $f \in \Theta(n^2)$. ■

Las definiciones clásicas se refieren a funciones monovariable, que solo tienen un argumento. Sin embargo, una UBF puede ser una función multivariable, esto es de varias variables naturales, bien porque el programa tenga más de una entrada [66, 65, 84], o bien porque la UBF se defina sobre varias medidas de tamaño de la entrada [73, 62, 6]. Dado que no hay una definición estándar para la notación asintótica de funciones multivariable, ni tampoco una demostración acerca de sus propiedades, es habitual que algunas veces se usen estas notaciones con funciones multivariable como si estas propiedades se mantuvieran. Sin embargo, Howell [77] demostró que algunas propiedades de las notaciones asintóticas, propiedades que son necesarias para construir una transformación o comparación asintótica automática, no se cumplen si las notaciones se generalizan a todas las funciones multivariable. Por suerte, de su trabajo se deducen dos condiciones para poder definir las notaciones asintóticas de tal manera que se preserven las propiedades cruciales:

[2] A veces se utiliza $f \preceq g$ y $f \approx g$ para abreviar $f \in O(g)$ y $f \in \Theta(g)$ [61, §4.1] [111, §9.1]

[3] Formalmente, para $0 \leq a \leq b$, se cumple que $n^a \in O(n^b)$, y para todo $1 < a \leq b$ se cumple que $a^n \in O(b^n)$.

- Las notaciones asintóticas se deben restringir a funciones multivariable que sean monótonas (no decrecientes) en todas las entradas; y
- las notaciones asintóticas deben definirse como una relación entre los valores de las funciones para vectores de entrada en el que todas las componentes de la entrada son grandes.

Por ello se definen las notaciones asintóticas para funciones multivariable [2, Definition 2] generalizando las definiciones para funciones univariable del siguiente modo: en vez de una variable n se considera un vector $\bar{n}$ de entrada. Así, si g es una función en $\mathbb{N}^m \mapsto \mathbb{R}^+$, $O(g)$ es el conjunto de funciones f in $\mathbb{N}^m \mapsto \mathbb{R}^+$ para las que existe algún número real $c_u > 0$ tal que la condición $f(\bar{n}) \leq c_u g(\bar{n})$ es cierta para vectores de entrada grandes. En este contexto se considera que un vector es grande si todas sus componentes lo son.

Ejemplo 10.2. Consideremos la función multivariable $f(A,B) = A * (2 * B + 3)$. Usando las propiedades anteriores se puede deducir que $2 * B + 3 \in \Theta(B)$, luego $f(A,B) \in \Theta(A * B)$. ■

Las expresiones de coste son funciones sobre variables enteras y no sobre naturales. Por ello, para utilizar las notaciones asintóticas previamente se tiene que transformarlas (o aproximarlas) a funciones multivariable, que además tienen que ser monótonas en cada componente. Para ello se debe identificar qué elementos de la expresión de coste se pueden abstraer a una variable natural de tal manera que la expresión sea monotónica en esos elementos. En nuestro caso, los elementos que pueden abstraerse son las subexpresiones nat (Sección 9.4). Esta elección se justifica con el siguiente ejemplo.

Ejemplo 10.3. Consideremos la expresión $\mathsf{nat}(l_0 - i_0) * (2 * \mathsf{nat}(l_0 - 1) + 3)$ del Ejemplo 9.7. Dado que ésta es monótona en sus componentes nat, se puede abstraer a la función $A * (2 * B + 3)$, donde A y B son variables naturales que abstraen $\mathsf{nat}(l_0 - i_0)$ y $\mathsf{nat}(l_0 - 1)$ respectivamente. No se puede abstraer esta expresión de coste abstrayendo l_0 e i_0 a variables naturales, pues la expresión no es monótona en i_0. ■

Ahora, definimos las notaciones para una expresión de coste e como una función multivariable f_e, donde cada argumento natural de f_e se corresponde con una subexpresión nat de e. Esta abstracción debe asociar subexpresiones nat distintas a variables distintas de f_e, pero si una subexpresión nat aparece varias veces en e, debe usarse la misma variable de f_e para cada aparición. Decimos que f_e es la nat-abstracción de exp, y que sus argumentos son sus variables nat.

Ejemplo 10.4. En el Ejemplo 10.3, para $e \equiv \mathsf{nat}(l_0 - i_0) * (2 * \mathsf{nat}(l_0 - 1) + 3)$, se usa la nat-abstracción $f_e \equiv A * (2 + B * 2 + 1)$. ■

La relación asintótica entre expresiones de coste se define como la relación entre sus abstracciones libres de nat. Es decir, si f_1 y f_2 son las nat-abstracciones de e_1 y e_2, entonces $e_1 \in O(e_2)$ si $f_1 \in O(f_2)$.

La definición de notación asintótica basada en el uso de la nat-abstracción está basada en intuiciones básicas del análisis de complejidad [15, 84, 65]. La complejidad asintótica de un programa está relacionada con el numero de iteraciones que cada bucle ejecuta. Como se vio en la Sección 9.5, el número de iteraciones de un bucle está relacionado con la altura de un árbol de evaluación, que a su vez está acotado por una expresión $\mathsf{nat}(l)$, donde l es una función de rango lineal. Así, las expresiones nat son las principales componentes que afectan al coste de un programa, por lo que se puede definir la complejidad de un programa como aquella la que las subexpresiones nat toman valores grandes.

2 Transformación Asintótica

En general, una transformación asintótica [126, 2] es un procedimiento automático que recibe como entrada una expresión de coste e y devuelve una expresión e' más simple y asintóticamente equivalente. Dicha transformación consiste en una secuencia de pasos para eliminar operadores y operandos redundantes: si a y b son dos expresiones y $\diamond$ es una operación, se dice que b (o $\diamond$) es redundante si $a \diamond b \in \Theta(a)$, esto es, si ni $\diamond$ ni b cambian la complejidad asintótica. En nuestro caso, una transformación asintótica toma como entrada una cota definida como un par $\{e, \varphi\}$, donde e es una expresión de coste y φ es una conjunción de restricciones lineales sobre las variables de e, a la que se llama como restricción de contexto. Nuestra transformación asintótica consiste en tres pasos:

1. **Calcular la** nat **abstracción** f_e descrita en la sección previa. Sin embargo, para no generar demasiadas variables nat, primero se transforma e de tal manera que las expresiones nat proporcionales se reemplacen por una forma común, abstrayéndose a la misma variable nat.
2. Se transforma la nat-abstracción f_e a una **forma normal asintótica** del siguiente modo: (1) cada operación $\max(e_1, e_2)$ se reemplaza por $e_1 + e_2$; (2) se quitan todas las constantes multiplicativas; (3) se reemplaza cada expresión de coste básica $b^A - 1$ por b^A; (4) se reemplaza cada expresión de coste logarítmica $\log_b(A+1)$ por $\log A$; y (5) se reescribe la expresión como una suma de productos de expresiones básicas de coste (Sección 9.4).
3. **Eliminar los términos redundantes** de la expresión de coste normalizada, que son aquellos sumandos asintóticamente menores que otros y que por tanto no afectan al orden asintótico de la suma.

Los dos primeros pasos son simplemente transformaciones asintóticas fáciles de implementar. Para el último paso es necesario usar una comparación asintótica que se describe a continuación.

Ejemplo 10.5. Consideremos la expresión $e = \mathsf{nat}(l_0) * (2 * \mathsf{nat}(l_0 - 1) + 3)$ del Ejemplo 9.7. Nuestra transformación procede así: (1) $\mathsf{nat}(l_0 - 1)$ se reemplaza por $\mathsf{nat}(l_0)$ y así se obtiene la nat-abstracción $f_e = A * (2 + A * 2 + 1)$; (2) se eliminan las constantes en f_e y se transforma a la forma normal $A^2 + A$; (3) se elimina el término redundante A para así obtener A^2. Para calcular la cota con las expresiones nat originales, deshacemos la nat-abstracción, generando $\mathsf{nat}(l_0)^2$. ■

3 Comparación Asintótica

Nuestro procedimiento de comparación asintótica recibe como entrada dos expresiones e_1 y e_2 que no contienen nat, junto con una restricción de contexto φ, e intenta demostrar que $e_1 \in O(e_2)$. Se basa en las nociones siguientes:

- **La subsunción asintótica**. Si A y B son nat-variables que corresponden a $\mathsf{nat}(l_1)$ and $\mathsf{nat}(l_2)$ respectivamente, se dice que A subsume a B (módulo φ) si φ implica que $\mathsf{nat}(l_1)$ es asintóticamente mayor que $\mathsf{nat}(l_2)$
- **El peso asintótico**. Es la medida clave para poder comparar el crecimiento asintótico de las expresiones. Por ejemplo, dicho crecimiento puede medirse obserbando la base de una expresión exponencial y el grado de un polinomio o polilogaritmo [4]. Así pues para comparar el peso de dos expresiones primero se compara la base exponencial, después el grado de los polinomios y finalmente el grado de los polilogaritmos.

Estas nociones ofrecen una vía directa para comparar dos expresiones de coste e_1 y e_2, módulo una restricción de contexto φ:

(R1) Para demostrar que $P \in O(b)$, siendo b una expresión de coste básica y P un producto de expresiones básicas, basta con probar (1) que cada nat-variable de b subsume (módulo φ) a cada nat-variable de P; y (2) que el peso de b es mayor que el de P.

(R2) Para demostrar que $P \in O(Q)$, siendo P y Q productos de expresiones de coste básicas, se intenta factorizar P en k subproductos $P = p_1 * p_2 * \ldots * p_k$, para los que existan k factores b_i distintos en Q tales que $p_i \in O(b_i)$.

(R3) Para demostrar que $S \in O(T)$, siendo S y T sumas de productos, solo tenemos que encontrar para cada sumando a de S un sumando a' en T tal que $a \in O(a')$.

Veamos un ejemplo de estas reglas.

[4] Un poli-logaritmo es una expresión como $(\log A)^2$.

Ejemplo 10.6. Comparamos las expresiones nat-abstractas $e_1 \equiv 2^B C^2 + A^3 D + D^2 A$ y $e_2 \equiv B^7 C + A^2 \log^2 B + B^2 C^2 + D^2 \log C$, siendo $A = \mathsf{nat}(x+y)$, $B = \mathsf{nat}(x)$, $C = \mathsf{nat}(y)$ y $D = \mathsf{nat}(z)$. Dada la restricción de contexto $\varphi \equiv \{x \geq 0, y \geq 0, z \geq y\}$, se pueden deducir las relaciones de subsunción $A \succcurlyeq B, A \succcurlyeq C, D \succcurlyeq C$. A partir de aquí se construye una prueba de $e_2 \in O(e_1)$ de este modo:

- Aplicando (R1) se deducen estas relaciones:

$$B^7 \in O\left(2^B\right) \quad A^2 \log^2 B \in O\left(A^3\right) \quad B^2 C \in O(A^3) \quad C \in O\left(C^2\right)$$
$$C \in O(D) \qquad \log C \in O(A) \qquad D^2 \in O(D^2)$$

  En el caso de $A^2 \log^2 B \in O(A^3)$, se tiene que A subsume tanto a A como a B, y el peso asintótico de A^3 es mayor que el de $A^2 \log^2 B$.
- Aplicando (R2) se deducen las siguientes relaciones:

$$B^7 C \in O(2^B C^2) \qquad B^2 C^2 \in O(A^3 D)$$
$$A^2 \log^2 B \in O(A^3 D) \quad D^2 \log C \in O(D^2 A)$$

  Por ejemplo, en el caso de $A^2 \log^2 B \in O(A^3 D)$, solo se usa un subproducto y factor ya que $A^2 \log^2 B \in O(A^3)$. En el de $B^2 C^2 \in O(A^3 D)$, se factoriza $B^2 C^2$ como $B^2 C * C$ y se usan las relaciones $B^2 C \in O(A^3)$ y $C \in O(D)$.
- Aplicando (R3) se deduce que

$$B^7 C + A^2 \log^2 B + B^2 C^2 + D^2 \log C \in O\left(2^B C^2 + A^3 D + D^2 A\right)$$

  porque para cada sumando en la izquierda hay uno en la derecha que es asintóticamente mayor, según las relaciones del paso anterior.

Así se concluye que $e_2 \in O(e_1)$. ■

El procedimiento de comparación anterior es correcto pero no completo, ya que existen casos en los que se cumple $e_1 \in O(e_2)$ y sin embargo el procedimiento no es capaz de demostrarlo.

Ejemplo 10.7. Para $e_1 \equiv A^2 + B^2$ y $e_2 \equiv AB$, se cumple que $e_2 \in O(e_1)$, pero nuestra comparación automáticamente falla. ■

4 Análisis Asintótico de Coste

En esta sección se discute cómo construir un analizador de coste asintótico que infiere directamente cotas asintóticas. Una solución trivial sería redirigir la salida de cualquier analizador de coste no asintótico [73, 66, 57, 90, 130] como entrada del proceso de transformación asintótica.

Ejemplo 10.8. Si se aplica la transformación asintótica a las UBFs del Ejemplo 9.7, se obtienen las siguientes UBFs asintóticas:

$$\begin{array}{ll} \mathit{inSort}(l_0) = \mathsf{nat}(l_0)^2 & \mathit{insert}(i_0) = \mathsf{nat}(i_0) \\ \mathit{for}(l_0, i_0) = \mathsf{nat}(l_0 - i_0) * \mathsf{nat}(l_0) & \mathit{while}_a(j_0) = \mathsf{nat}(j_0) \end{array}$$

que son precisamente las que se obtienen analizando manualmente el coste asintótico de los métodos y bucles del programa `inSort` de la Figura 9.2. ■

Esta solución es claramente correcta pero ineficiente, debido a que el analizador de coste genera información detallada que es desechada por la transformación asintótica. Para evitar la generación de dicha información, hemos desarrollado un **resolutor asintótico de CRSs**, que genera directamente cotas asintóticas. Este método es más eficiente, aunque solo se puede aplicar a analizadores de coste basados en el enfoque clásico. El resolutor asintótico sigue las fases de PUBS (véase Sección 9.5), salvo que se aplica la transformación asintótica siempre que se genera una expresión no asintótica. En concreto, se aplica en los siguientes pasos de PUBS:

- Al transformar el CRS a recursión directa, PUBS genera complejas ecuaciones de coste al sumar aquéllas de las ecuaciones desplegadas. Se usa la transformación asintótica para reducir estas expresiones.
- Para transformar cada CR en el CRS en una CR aislada, PUBS reemplaza cada llamada externa por la UBF de la CR llamada, y añade la UBF a la expresión de coste de la ecuación. En ese punto se aplica la transformación.
- Para calcular una UBF de una CR aislada, antes de almacenar esta UBF se aplica la transformación.

Los métodos para calcular una función de rango, para maximizar las expresiones, y para resolver una CR, permanecen igual que en PUBS.

Ejemplo 10.9. Aplicamos la resolución asintótica al CRS de la Figura 9.6. El CRS se transforma a recursión directa como en el ejemplo 9.6, salvo que al generar la tercera ecuación de while_a la transformación asintótica reemplaza la constante 2 por 1; en las demás ecuaciones la expresión de coste ya está en forma asintótica. El siguiente paso es resolver cada CR, como se hace en ejemplo 9.7:

- La UBF de la CR $\mathit{while}_a(j_0)$ es $\mathsf{nat}(j_0) + 1$. La transformación lo reduce a la UBF asintótica $\mathit{while}_a{}^+(j_0) = \mathsf{nat}(j_0)$.
- Al sustituir $\mathit{while}_a{}^+(j_0)$ en la CR de *insert*, se obtiene una CR aislada con una ecuación cuya expresión de coste es $1 + \mathsf{nat}(i_0)$, que aplicando la transformación se reduce a $\mathsf{nat}(i_0)$, obteniéndose la AUBF $\mathit{insert}^+(i_0) = \mathsf{nat}(i_0)$.
- Al sustituir $\mathit{insert}^+(i_0)$ en la CR *for*, se obtiene una CR aislada cuya ecuación recursiva tiene la expresión de coste $\mathsf{nat}(i_0)$, ya en forma asintótica. Al resolver esta CR se obtiene la UBF $\mathsf{nat}(l_0 - i_0) * \mathsf{nat}(l_0 - 1)$, que la transformación asintótica reduce a $\mathit{for}^+(l_0, i_0) = \mathsf{nat}(l_0 - i_0) * \mathsf{nat}(l_0)$.

- Al sustituir $for^+(l_0, i_0)$ en la CR *inSort*, y resolverla, se obtiene la UBF $\mathsf{nat}(l_0 - 1) * \mathsf{nat}(l_0)$, que la transformación asintótica reduce a $inSort^+(l_0) = \mathsf{nat}(l_0)^2$.

Nótese que estas AUBFs son iguales a las del ejemplo 10.8. ■

5 Evaluación Experimental

En esta sección se describe una evaluación experimental de nuestras técnicas. Como no hay otra técnica que calcule cotas asintóticas, hemos comparado nuestras técnicas con ésas de PUBS para inferir cotas no asintóticas.

Evaluación de la Transformación Asintótica. Hemos implementado nuestra transformación como un *back-end* de COSTA. Para evaluarla, se la aplicó a las UBFs de consumo de memoria que se obtienen en [13, §7]. Para todos los programas, la transformación obtiene una forma asintótica mínima y precisa, en un tiempo insignificante. Esto demuestra que, con nuestra transformación, todo el trabajo existente para calcular cotas no asintóticas se puede aplicar para calcular cotas asintóticas. Esta técnica se puede aplicar para UBFs y LBFs, con cualquier modelo de coste y medida de tamaño.

Evaluación de la Resolución Asintótica. En otra serie de experimentos se estudia la escalabilidad de nuestro enfoque, esto es, cómo aumenta tanto el tamaño de las UBFs como el tiempo que se tarda en calcularlas con respecto al tamaño del CRS. Para ello se usan los mismos *benchmarks* de escalabilidad que se usan en [5, §10.2]. Para cada uno, se calculó (usando PUBS) tanto una UBF como una AUBF, y se vio que: (1) el tiempo para calcular una UBF crece de manera significativa con el tamaño de las CRs, mientras que el tiempo para calcular una AUBF permanece menor; (2) en cada ejemplo la UBF es significativamente más grande que la AUBF; (3) la entre el tamaño de las cotas calculadas y el tamaño (número de ecuaciones) del CR crece más rápido para las UBFs que para las AUBFs; (4) para los *benchmarks* más grandes, no se pudo calcular la UBF no asintótica en un tiempo razonable, pero sí la AUBF. Estas observaciones muestran que nuestro enfoque es más escalable.

6 Trabajo Relacionado

Notaciones asintóticas. Knuth [88] definió formalmente las notaciones asintóticas para funciones monovariable. Algunos autores [45, §3.1],[111, §9.2] usan esta definición porque es fácil de traducir a una fórmula lógica con la que demostrar las propiedades básicas. Otros trabajos [118, §1.2], [61, §4.1.1], usan una definición equivalente de O y Θ, basada en el límite del cociente $f(n)/g(n)$

cuando n tiende a infinito. Ésta definición es más intuitiva, ya que transmite la intuición de que se estudia la proporción entre funciones para entradas arbitrariamente grandes. Nuestro trabajo se centra en las notaciones big-O y big-Theta, pero se puede extender a las notaciones big-Omega Ω, little-o $o(f)$ y little-omega $\omega(f)$.

Una parte fundamental de nuestro trabajo ha sido extender estas notaciones a funciones multivariable y, al mismo tiempo, preservar las propiedades algebraicas correspondientes [77]. Hemos resuelto este problema restringiendo las definiciones a funciones que son monótonas en todas las entradas.

Transformación y Comparación Asintótica. Stoutemyer [126] presenta otro método para la transformación y comparación asintótica de funciones. Su método está construido sobre el sistema MACSYMA de álgebra por computador, y usa algunas de sus características avanzadas, las cuales cubren un conjunto muy amplio de expresiones. En cambio, nuestro método puede manejar información de contexto dada como restricciones lineales sobre las variables. Nuestro método de comparación guarda semejanzas con el método de PUBS para comparar expresiones no asintóticas de coste [4].

Análisis Asintótico de Coste. Las primeros trabajos en análisis estático de coste [41, 43] distinguían entre macro- y micro- análisis de programas, donde el primero se centra en una operación dominante y el segundo considera todas las operaciones. Bajo esta visión, una AUBF es un macroanálisis. En [101], se presenta un análisis automático de complejidad para cláusulas de Horn. El sistema ACE [96] realiza un análisis de coste asintótico, basado en la transformación de programas, para programas escritos en un lenguaje funcional de primer orden. El análisis de complejidad, o el análisis de complejidad asintótica, también se ha desarrollado para lenguajes imperativos [84], sistemas de reescritura [102], y a traves de éstos para lenguajes de programación lógica [59].

7 Contenido Adicional

En este capítulo se han descrito las principales contribuciones del artículo [2], que son (1) la extensión de las notaciones asintóticas a las expresiones de coste; (2) una transformación y comparación asintótica para expresiones de coste; y (3) un resolutor asintótico de CRSs. Cada contribución se ha ilustrado con ejemplos sencillos. En el artículo se definen formalmente la notación asintótica, así como la transformación y comparación asintótica para expresiones de coste. También se proporcionan resultados matemáticos sobre la corrección de éstos métodos, cuyas demostraciones pueden encontrarse en el apéndice.

11 | Operaciones No Lineales

En este capítulo se describe un análisis de valores para operaciones aritméticas no lineales sobre enteros. Con él, COSTA infiere UBFs precisas para programas para los cuales COSTA antes fallaba o infería una UBF imprecisa. Esta contribución se publicó en el siguiente artículo:

> DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Handling Non-linear Operations in the Value Analysis of COSTA. *Electronic Notes in Theoretical Computer Science*, 279(1):3–17, 2011. Proceedings of Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)

el cual se halla anexado en esta disertación, en la Página 217.

Descripción

El objetivo principal de esta tesis es mejorar la precisión de COSTA, para que calcule UBFs y LBFs precisas para una amplia clase de programas. Este Capítulo se centra en los componentes que abstraen las instrucciones RBR a restricciones lineales, infieren para cada procedimiento del ACR una postcondición que también es un conjunto de restricciones lineales, y añaden ambas al CRS. Nuestro objetivo es mejorar la precisión de estos componentes, para que así calculen una aproximación más ajustada, esto es, restricciones más fuertes. La fuerza de estas restricciones tiene un efecto directo en la precisión de las costas inferidas: si ésas son demasiado débiles, la semántica del CRS incluye evaluaciones espúreas, esto es, que no corresponden a ninguna traza de la RBR.

Las restricciones del CRS aproximan los valores de las variables del programa en diferentes puntos del programa, así como las relaciones entre esos valores. Como se explica en el Capítulo 9, en COSTA estas restricciones se generan en la compilación abstracta y en el análisis de valores. Estos componentes se basan en la teoría de Interpretación Abstracta *(Abstract Interpretation)* [48, 46]. Esta teoría parte del hecho de que no se puede calcular el conjunto exacto de relaciones de valores. En vez de ello, en una interpretación abstracta se usa un **dominio abstracto**, esto es una clase sintáctica de formulas lógicas [34, §12.1.4], el cual restringe cómo se aproxima el efecto de cada instrucción y cómo se representan las relaciones de valor que se infieren.

En COSTA se usan conjunciones de restricciones lineales. Esto sirve para analizar con precisión una clase amplia de programas, pero que no bastan para

```
1   static int clog(int x, int b){
2       if (b > 1) {
3           int y = 1 ;
4           int z = 0 ;
5           while ( y < x ){
6               y = y * b ;    ★
7               z = z + 1 ;
8           } ;
9           return z ;
10      } else  return 0 ;
11  }
```

Figura 11.1: Código JAVA para el ejemplo `clog`. La ★ señala el producto.

modelar la semántica de las operaciones no lineales, como el producto $z = x * y$. Si el coste de un programa depende de una instrucción no lineal, esto afecta a la precisión de las cotas inferidas.

En este capítulo se trata la imprecisión causada por esas instrucciones aritméticas no lineales. La idea clave es que, si bien no se puede modelar una operación de éstas con *una* conjunción de restricciones, sí se puede con una *disyunción* finita de conjunciones, dividiendo la semántica de una operación no lineal en varios casos. Si estas disyunciones se codifican como procedimientos del programa ACR, se puede usar un análisis de valor escalable que pueda aprovechar esa información para inferir relaciones de valor precisas.

En la Sección 11.1 se explica el funcionamiento interno de nuestro método. En la Sección 11.3 se menciona trabajo relacionado, y en la Sección 11.4 se describe el contenido adicional del artículo.

1 Manejar una Operación No Lineal

La compilación abstracta de COSTA abstrae las operaciones no lineales a la restricción *true*, que representa el espacio de todos los estados del programa. Ello provoca una pérdida de precisión que lleva a inferir una UBF imprecisa, como en el siguiente ejemplo.

Ejemplo 11.1. La Figura 11.1 muestra un método JAVA `clog`, que toma como entrada dos números positivos `x` y `b`, y devuelve el valor de $\lceil \log_b(x) \rceil$. Se quiere analizar el coste de este programa con respecto a un modelo de coste que cuenta, por ejemplo, el número de visitas al cuerpo del bucle. Para este ejemplo, la

$clog(\langle x, b\rangle, \langle r\rangle) \leftarrow$
  $if_c(\langle x, b\rangle, \langle r\rangle)$.
$if_c(\langle x, b\rangle, \langle r\rangle) \leftarrow$
  $\{\neg b > 1\}$,
  $r := 0$.
$if_c(\langle x, b\rangle, \langle r\rangle) \leftarrow$
  $\{b > 1\}$,
  $y := 1$,
  $z := 0$,
  $while_c(\langle x, b, y, z\rangle, \langle z\rangle)$,
  $r := z$.
$while_c(\langle x, b, y, z\rangle, \langle z\rangle) \leftarrow$
  $\{\neg y < x\}$.
$while_c(\langle x, b, y, z\rangle, \langle z\rangle) \leftarrow$
  $\{y < x\}$,
  $y := y * b$, ★
  $z := z + 1$,
  $while_c(\langle x, b, y, z\rangle, \langle z\rangle)$.

$clog(\langle x_0, b_0\rangle, \langle\rangle) \leftarrow$
  $if_c(\langle x_0, b_0\rangle, \langle\rangle)$.
$if_c(\langle x_0, b_0\rangle, \langle\rangle) \leftarrow$
  $b_0 \leq 1$.

$if_c(\langle x_0, b_0\rangle, \langle\rangle) \leftarrow$
  $b_0 \geq 2$,
  $y_0 = 1$,
  $while_c(\langle x_0, b_0, y_0\rangle, \langle\rangle)$.

$while_c(\langle x_0, b_0, y_0\rangle, \langle\rangle) \leftarrow$
  $y_0 \geq x_0$.
$while_c(\langle x_0, b_0, y_0\rangle, \langle\rangle) \leftarrow$
  $y_0 + 1 \leq x_0$,
  $true$, ★
  acquire(1),
  $while_c(\langle x_0, b_0, y_1\rangle, \langle\rangle)$.

Figura 11.2: La RBR (izquierda) y la ACR (derecha) que COSTA obtiene para el ejemplo `clog`. La ★ señala la operación no lineal.

UBF precisa está en $\Theta(\log(x))$. La Figura 11.2 muestra los correspondientes programas RBR (izquierda) y ACR (derecha) generados por COSTA. Las variables r y z de la RBR no están en la ACR porque no afectan al coste. La pérdida de información se produce en la compilación abstracta. En el método JAVA (y en el programa RBR), la instrucción `y = y * b` (resp. $y := y * b$) asigna a `y` (resp. y) el valor de `y * b` (resp. $y * b$). Sin embargo, la restricción $\top$ en el programa ACR significa que la variable y_1 puede tomar cualquier valor, por lo que el coste de este programa ACR es ilimitado. ■

Para resolver el problema de precisión e inferir una UBF precisa para el método `clog`, se debe manejar la operación del producto de manera más precisa. Esto es, se quiere mejorar el análisis de valores para que obtenga una relación más precisa para esa instrucción. Para ello, se podría simplemente cambiar el dominio abstracto de los poliedros por uno que representa relaciones no lineales [115, 28, 63], pero este análisis de valor sería poco útil ya que las operaciones en estos dominios no son escalables.

Nuestra solución parte de esta observación: aunque la instrucción no sea lineal, si se tiene información de contexto se la puede aproximar por una relación lineal. Esto es, si una restricción lineal (de contexto) se cumple sobre

los operandos, entonces puede que haya una restricción lineal no trivial que se cumpla sobre el resultado.

Ejemplo 11.2. Observe la instrucción $y := y * b$ en la RBR de la Figura 11.2. Si no se sabe nada sobre el valor de y y b, el único modo en que la semántica de $y := y * b$ se puede abstraer a una conjunción de restricciones lineales es a usando la restricción *true*. Ahora bien, si se sabe que la restricción $\{y \geq 1, b \geq 2\}$ se cumple antes de la instrucción, entonces se puede usar esta información de contexto para afinar la abstracción de $y := y * b$ en $\{y_1 \geq 2 * y_0\}$, siendo y_0 e y_1 los valores de y antes y después de la instrucción. ■

La esencia de nuestra solución es usar la información sobre el contexto de la operación no lineal para refinar la abstracción. Para ello, cada instrucción se abstrae a una disyunción de casos, donde en cada caso se especifica un escenario de entrada usando una conjunción de restricciones lineales. Para que la abstracción sea correcta, los escenarios de los casos deben cubrir todo el dominio de entrada. El reto es cómo representar dichas disyunciones manteniendo la el análisis de valores escalable.

Una solución directa es utilizar un dominio abstracto de *powerset*, los cuales representan directamente disyunciones finitas, pero tales dominios dañan mucho el rendimiento, lo que vuelve el análisis poco útil. Además, se puede observar que la información disyuntiva no se necesita en el análisis global del programa, sino más bien localmente para analizar el efecto de cada instrucción no lineal. Por ello, nuestra solución inspirada en [114] es codificar las disyunciones directamente en la ACR, sin usar restricciones disyuntivas, aprovechando la naturaleza disyuntiva de las reglas de un procedimiento de la ACR. Por ejemplo, la instrucción aritmética no lineal $x := e_1 * e_2$ se abstrae a una llamada a $op_*(\langle e_1, e_2\rangle, \langle x\rangle)$, y se define el procedimiento abstracto auxiliar op_* con varias reglas que cubren todas las posibles entradas, lo cual simula la disyunción deseada. Es importante señalar que cada regla de op_* solo usa una conjunción de restricciones lineales.

Ejemplo 11.3. La Figura 11.3 muestra el programa ACR obtenido al aplicar la nueva compilación abstracta al programa RBR del método `clog`. Este ACR es como el de la Figura 11.2, salvo que ahora la instrucción $y := y * b$ se abstrae a una llamada $op_*(\langle y_0, b_0\rangle, \langle y_1\rangle)$, y este procedimiento se define en el ACR. Las reglas de op_* dividen el dominio de entrada de tal manera que en cada regla la postcondición da información ajustada sobre el valor de salida. En concreto, se separan los casos en que $x = 0$ (constante), $x = \pm 1$ (igualdad) y aquellos en que $|x| > 1$ y $|y| > 1$ (progresión). Observe que en el caso (g) la postcondición $z \geq 2x$ y $z \geq 2y$ es crucial para inferir una UBF logarítmica para nuestro ejemplo. ■

$$
\begin{array}{l}
clog(\langle x_0, b_0\rangle, \langle\rangle) \leftarrow \\
\quad if_c(\langle x_0, b_0\rangle, \langle\rangle).
\end{array}
\qquad
\begin{array}{l}
while_c(\langle x_0, b_0, y_0\rangle, \langle\rangle) \leftarrow \\
\quad y_0 \geq x_0.
\end{array}
$$

$$
\begin{array}{l}
if_c(\langle x_0, b_0\rangle, \langle\rangle) \leftarrow \\
\quad b_0 \leq 1.
\end{array}
\qquad
\begin{array}{l}
while_c(\langle x_0, b_0, y_0\rangle, \langle\rangle) \leftarrow \\
\quad y_0 + 1 \leq x_0, \\
\quad \mathsf{acquire}(1), \\
\quad op_*(\langle y_0, b_0\rangle, \langle y_1\rangle, \quad \bigstar \\
\quad while_c(\langle x_0, b_0, y_1\rangle, \langle\rangle).
\end{array}
$$

$$
\begin{array}{l}
if_c(\langle x_0, b_0\rangle, \langle\rangle) \leftarrow \\
\quad b_0 \geq 2, \\
\quad y_0 = 1, \\
\quad while_c(\langle x_0, b_0, y_0\rangle, \langle\rangle).
\end{array}
$$

$$
\begin{array}{lll}
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow & & \\
\quad x = 0, & (a) & \\
\quad z = 0. & & \\
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow & & \\
\quad y = 0, & (b) & \\
\quad z = 0. & & \\
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow & & \\
\quad x = 1, & (c) & \\
\quad z = y. & & \\
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow & & \\
\quad y = 1, & (d) & \\
\quad z = x. & & \\
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow & & \\
\quad x = -1, & (e) & \\
\quad z = -y. & &
\end{array}
\qquad
\begin{array}{lll}
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow & & \\
\quad y = -1, & (f) & \\
\quad z = -x. & & \\
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow & & \\
\quad x \geq 2, y \geq 2, & (g) & \\
\quad z \geq 2x, z \geq 2y. & & \\
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow & & \\
\quad x \geq 2, y \leq -2, & (h) & \\
\quad z \leq -2x, z \leq 2y. & & \\
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow & & \\
\quad x \leq -2, y \geq 2, & (i) & \\
\quad z \leq 2x, z \leq -2y. & & \\
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow & & \\
\quad x \leq -2, y \leq -2, & (j) & \\
\quad z \geq -2x, z \geq -2y. & &
\end{array}
$$

Figura 11.3: Programa ACR para el método `clog` que se obtiene por nuestra compilación abstracta modificada. Ésta abstrae la instrucción $y := y * b$ a $op_*(\langle y_0, b_0\rangle, \langle y_1\rangle)$. Debajo, las reglas del procedimiento auxiliar op_*.

Tras la compilación abstracta, que abstrae cada operación no lineal a un procedimiento auxiliar, el análisis de valores infiere qué casos de éste están activos en el contexto de la llamada. Para ello, se aplica una interpretación abstracta que infiere una precondición que describe cómo se llama a ese procedimiento ACR. Técnicamente, esto se hace usando un algoritmo que calcula el menor punto fijo (*least fixed point* o LFP) de la semántica de recolección abstracta. Para mantener el análisis eficiente, usamos el dominio abstracto no disyuntivo de los poliedros. Con esa precondición, se obtiene una postcondición más precisa para la operación no lineal, como la menor cota superior (*least upper bound* o lub) de los casos que están activos. Una vez las pre y postcondiciones se han calculado, se genera el CRS tal y como en la Sección 9.4, salvo que también se añaden las precondiciones en el CRS. Luego, PUBS resuelve resolver

$$
\begin{array}{rll}
clog(x_0, b_0) = & if_c(x_0, b_0) & \{\} \\
if_c(x_0, b_0) = & 0 & \{b_0 \leq 1\} \\
if_c(x_0, b_0) = & while_c(x_0, b_0, y_0) & \{b_0 \geq 2, y_0 = 1\} \\
while_c(x_0, b_0, y_0) = & 0 & \{y_0 \geq x_0\} \\
while_c(x_0, b_0, y_0) = & 1 + while_c(x_0, b_0, y_1) & \{y_0 + 1 \leq x_0, \underbrace{b_0 \geq 2, y_0 \geq 1}_{\text{pre}-op_*}, \underbrace{y_1 \geq 2 * y_0}_{\text{post}-op_*}\}
\end{array}
$$

Figura 11.4: CRS inferido para el método `clog` con nuestro análisis de valores.

este CRS en una UBF igual que en la Sección 9.5.

Ejemplo 11.4. Considere el programa ACR en la Figura 11.3. Para el procedimiento $op_*(\langle b_0, y_0\rangle, \langle y_1\rangle)$, se infiere la precondición $\varphi \equiv \{b_0 \geq 2, y_0 \geq 1\}$. Para cada regla de op_*, se calcula la conjunción de φ con las restricciones en la regla: la regla está activa si esa conjunción es distinto de *false*. Solo las reglas (d) y (g) están activas, siendo $\varphi_d \equiv \{y_0 \geq 2, b_0 \geq 1, y_1 = y_0\}$ y $\varphi_g \equiv \{y_0 \geq 2, b_0 \geq 2, y_1 \geq 2y_0, y_1 \geq 2b_0\}$ sus respectivas conjunciones. La postcondición de $op_*(\langle b_0, y_0\rangle, \langle y_1\rangle)$ se calcula como el lub (*convex hull*) de φ_d y φ_g, que es $\varphi' \equiv \varphi \wedge \{y_1 \geq 2 * y_0\}$. A partir del programa ACR en la Figura 11.3, COSTA genera el CRS en la Figura 11.4. Observe que la pre y postcondición de op_* se añaden en la segunda ecuación de la CR $while_c$. Después, PUBS resuelve este CRS en la UBF $clog(x_0, b_0) = \log_2(\mathsf{nat}(x_0) + 1))$, que es asintóticamente precisa. ■

A diferencia del análisis de valores aplicado al programa ACR en COSTA, que sigue una estrategia ascendente [30], para que nuestro análisis pueda manejar instrucciones no lineales se necesita una estrategia descendente [40], la cual también infiere precondiciones. Por último, hay que señalar que en nuestra abstracción se ignora el desbordamiento aritmético, pues asumimos que éste es un comportamiento erróneo que debe tratarse aparte.

2 Evaluación Experimental

Hemos implementado el análisis de valor en COSTA, para lo cual (1) se ha implementado el algoritmo de LFP descendente que se describe en el artículo; (2) se ha modificado la compilación abstracta de las operaciones RBR no lineales, para que introduzca en la ACR los procedimientos auxiliares. Como implementación del dominio abstracto, se usa la *Parma Polyhedra Library* [24].

Para nuestros experimentos, se usan programas que usan operaciones no lineales y de vectores de bits. Éstos vienen de la literatura en análisis de programas, y algunos de las bibliotecas estándar de JAVA.

Para cada programa, se analizó su terminación y su coste (con el modelo de coste del número de instrucciones) con COSTA, con y sin nuestra extensión. Tal

y como se esperaba, sin nuestra extensión COSTA falló en todos los programas, mientras que con nuestra extensión logró demostrar la terminación e inferir un UBF para cada programa. Nuesto análisis de valor es más lento que el de COSTA, si bien esto se debe a que se usa un algoritmo LFP descendente, en vez de uno ascendente [10].

También probamos otros dos analizadores de terminación para JBC, JULIA [123] y APROVE [60], sobre los mismos programas. JULIA no pudo probar la terminación en ninguno de ellos. APROVE no pudo analizar los programas que usaban operaciones de vectores de bit, pero sí pudo probar la terminación de aquéllos que usan operaciones aritméticas no lineales; no obstante, el tiempo que tardó era significativamente (un orden de magnitud) mayor.

Con estos experimentos se confirma que, al usar un análisis de valores que 1) abstrae las operaciones no lineales a disyunciones de casos, 2) codifica dichas disyunciones como reglas adicionales en el programa ACR, 3) usa un dominio abstracto no disyuntivo, como los polihedros, y 4) emplea un algoritmo LFP descendente; se puede mejorar la precisión del análisis de valores para programas que usan operaciones no lineales.

3 Trabajo Relacionado

El análisis de valores de COSTA se basa en **Interpretación Abstracta** [48, 46], una teoría para el análisis semántico de programas, y para la derivación sistemática de analizadores estáticos. Esta teoría se ha usado para desarrollar analizadores industriales, como ASTRÉE [49] y Julia [123].

Compilación Abstracta. La idea de *compilación abstracta* aparece primero en [70] como una transformación sintáctica del programa en un programa abstracto, el cual luego se analiza para deducir una propiedad de interés. Boucher [33] presenta la compilación abstracta, extendiéndolo de una transformación sintáctica con optimizaciones sobre el programa abstracto.

Dominios Abstractos Numéricos. Además del dominio de los poliedros [67], en el algoritmo de LFP de nuestro método se puede usar un dominio abstracto débilmente relacional [25] para representar las desigualdades lineales. La ventaja de hacerlo es que, al restringir la forma de las restricciones lineales, las operaciones abstractas para estos dominios son más baratas computacionalmente que las de los poliedros, por lo que el algoritmo de LFP escala mejor. La desventaja es que al hacerlo se pierde precisión: Por ejemplo, en algunos de estos dominios no se puede representar la desigualdad $y_1 \geq 2 * y_0$ del Ejemplo 11.4, que hace falta para inferir una AUBF precisa.

Se podría representar algunas instrucciones aritméticas no lineales, como el producto, usando un dominio abstracto de conjunciones de igualdades [110] y desigualdades [28] entre polinomios multivariable de grado limitado. El método de [63] convierte un dominio abstracto de desigualdades lineales hacia uno de restricciones entre expresiones logarítmicas, exponenciales, radicales o con máx. Sin embargo, estos dominios son computacionalmente caros.

Se podría usar el dominio abstracto de restricciones lineales entre valores *absolutos* [37] para abstraer algunas operaciones no lineales a una restricción no trivial. Por ejemplo, se podría abstraer $\mathtt{z} := \mathtt{x}\,\%\mathtt{y}$ como $|z| < |y| \wedge |z| \leq |x|$.

Interpretación Abstracta con Información Disyuntiva. En nuestro método aparece el problema de cómo manejar información disyuntiva en una interpretación abstracta, para lo cual existen varias soluciones. Por un lado, hay dominios abstractos *powerset* [24], que representan disyunciones finitas. Sin embargo, en [114] se expone que el análisis de valor con estos dominios no es escalable. Como alternativa, se propone manejar la información disyuntiva con una transformación del programa, llamada elaboración, cuya idea básica es asociar cada nodo en que tenga un invariante disyuntivo en el CFG original con varios nodos del CFG elaborado, cuyos invariantes sean los elementos de la disyunción del invariante del primero. Otra técnica para manejar información disyuntiva es usar la partición de trazas *(trace partitioning)* [108], que divide el conjunto de trazas en la semántica de un programa en varios subconjuntos, cada uno de los cuales admite un invariante no disyuntivo.

4 Contenido Adicional

En este capítulo se ha descrito brevemente cómo manejar las operaciones no lineales en la compilación abstracta y el análisis de valor de COSTA, para inferir UBFs precisas para programas que usen esas operaciones. Sin embargo, en el artículo este análisis se orienta al análisis de terminación mediante funciones de rango.

En el artículo, se describe cómo aplicar el método para manejar, no solo el producto $\mathtt{z} = \mathtt{x} * \mathtt{y}$ sino también las operaciones de cociente entero $\mathtt{z} = \mathtt{x}/\mathtt{y}$ resto entero $\mathtt{z} = \mathtt{x}\,\%\mathtt{y}$, y las operaciones de vectores de bit como la *and* bit-a-bit $\mathtt{z} = \mathtt{x}\&\mathtt{y}$, la *or* bit-a-bit $\mathtt{z} = \mathtt{x}|\mathtt{y}$, desplazamiento a la izquierda $\mathtt{z} = \mathtt{x} << \mathtt{y}$, y desplazamiento a la derecha $\mathtt{z} = \mathtt{x} >> \mathtt{y}$. Para inferir una pre- y postcondición de cada procedimiento auxiliar, en el artículo se describe el algoritmo de LFP interprocedural, que sigue una estrategia descendiente [40].

12 | Amortización y Más Allá

Esta capítulo explora los límites del enfoque clásico de análisis de coste, que COSTA sigue. Este enfoque a veces infiere UBFs asintóticamente imprecisas, y se creía que la causa estaba en el método para resolver los CRSs en UBFs. Aquí mostramos que esto no es del todo cierto, e identificamos por qué razón el coste de algunos programas no se puede representar en un CRS de manera precisa. Para superar estas limitaciones, desarrollamos un enfoque de análisis de coste, basado en usar de *satisfiability modulo theory* (o SMT) y eliminación de cuantificadores (*quantifier elimination* o QE). Nuestro enfoque se desarrolla en un contexto en el que los programas, además de adquirir coste también lo pueden devolver, lo que da origen a la noción de coste pico. Nuestros resultados tienen una fuerte relación con el análisis amortizado de coste. Esta contribución se publicó en este artículo:

> DIEGO ESTEBAN ALONSO-BLAS AND SAMIR GENAIM. On the Limits of the Classical Approach to Cost Analysis. In Antoine Miné and David Schmidt, editors of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 7460 of *Lecture Notes in Computer Science*, pages 405–421. Springer, September 2012

el cual figura en la Página 233.

Descripción

El objetivo principal de esta tesis es mejorar la precisión de COSTA. Este capítulo se centra en la precisión del enfoque clásico en que se basa COSTA. Una idea básica de este enfoque es que la entrada de un procedimiento ACR determina la salida y el coste. Ello se refleja en que cada procedimiento ACR se abstrae en: (1) una CR que captura cómo el coste depende de la entrada, y (2) una postcondición de valores que aproxima la relación entre la entrada y la salida. En la Sección 12.1 se muestra que esta idea es la raíz de algunos problemas de imprecisión de COSTA. Para ello, se muestra un ejemplo en el cual hay una codependencia implícita entre la salida y el coste de un procedimiento de la ACR. Esta codependencia, que es crucial para inferir UBFs precisas, se pierde al transformar el ACR en el CRS, lo cual introduce en el CRS evaluaciones espúreas, que no existen en el ACR, por lo cual se infiere una UBF imprecisa.

Esta imprecisión se resuelve transformando esa codependencia implícita en una dependencia explícita, al añadir los parámetros de salida en la UBFs. A

```
1   //@requires m >= 0 && size >= 0
2   void main(int size , int m) {
3       for( ; m > 0 ; m-- ) {
4           while( size > 0  && coin() )
5               size = size - 1 ;          ★
6       }
7   }
```

Figura 12.1: Código JAVA para nuestro ejemplo guía.

estas UBFs se las llama como UBFs en el **coste neto**, porque describen el coste de las ejecuciones completas finitas. En la Sección 12.2 se describe una técnica para inferir tales UBFs, basada en QE y SMT.

En la Sección 12.3 se considera la noción de **coste pico**, que estima cuántos recursos puede como máximo retener un programa. Esta noción es útil tanto para modelos de coste no acumulativos, que representan recursos que se adquieren y se liberan, como para acotar el coste de programas no terminantes. Se desarrolla una técnica para inferir UBFs en el **coste pico**, también basada en QE y SMT. En la Sección 12.4 se discute la relación entre nuestro enfoque y el de análisis amortizado [79], y se muestra cómo las funciones de potencial de este enfoque se pueden ver como un caso restringido de UBFs en el coste neto. En la Sección 12.5 se discute una evaluación experimental de nuestro enfoque. En la Sección 12.6 se repasa el trabajo relacionado, y en la Sección 12.7 se describe el contenido adicional del artículo.

1 La Codependencia entre Salida y Coste

Para empezar, se muestra un ejemplo, adaptado de [45, §17], que revela el problema que queremos resolver, sobre la imprecisión del enfoque clásico.

Ejemplo 12.1. La Figura 12.1 muestra un método JAVA en el cual el bucle `for` externo realiza `m` iteraciones, y en cada una el bucle interno `while` decrementa la variable `size` un número arbitrario de veces, donde el método `coin` devuelve `true` o `false` aleatoriamente. Queremos inferir una UBF para el número de visitas a la Línea 5 (señalada con ★). Una UBF precisa está en $\Theta(\texttt{size})$.

Con COSTA, se genera el programa ACR en la Figura 12.2 y el CRS de la Figura 12.3. Para abreviar, se omite el programa RBR, así como el procedimiento del método *main*. En el programa ACR, la variable de salida s_1 de *while* representa el valor de `size` a la salida del bucle `while`. Al transformar la Regla (*b*) del ACR en la Ecuación (*b*) en el CRS, se quita el parámetro de salida s_2 de la llamada

(a) $for(\langle s_0, m_0\rangle, \langle\rangle) \leftarrow$
$m_0 = 0,$
$s_0 \geq 0.$

(b) $for(\langle s_0, m_0\rangle, \langle\rangle) \leftarrow$
$m_0 \geq 1,$
$s_0 \geq 0,$
$\underline{while(\langle s_0\rangle, \langle s_2\rangle)},$
$m_2 = m_0 - 1,$
$for(\langle s_2, m_2\rangle, \langle\rangle).$

(c) $while(\langle s_0\rangle, \langle s_1\rangle) \leftarrow$
$s_0 \geq 0,$
$s_1 = s_0.$

(d) $while(\langle s_0\rangle, \langle s_1\rangle) \leftarrow$
$s_0 \geq 1,$
acquire(1),
$s_2 = s_0 - 1,$
$while(\langle s_2\rangle, \langle s_1\rangle).$

Figura 12.2: Programa ACR para el ejemplo guía.

(a)	$for(s_0, m_0) = 0$	$\{m{=}0, s_0 \geq 0\}$
(b)	$for(s_0, m_0) = \underline{while(s_0)} + for(s_2, m_2)$	$\{m_0 \geq 1, \underline{s_0 \geq s_2 \geq 0}, m_2 = m_0{-}1\}$
(c)	$while(s_0) = 0$	$\{s_0 \geq 0\}$
(d)	$while(s_0) = 1{+}while(s_2)$	$\{s_0 \geq 1, s_2 = s_0 - 1\}$

Figura 12.3: CRS para al ejemplo guía.

a *while* y se añade la postcondición $s_0 \geq s_2 \geq 0$ en las restricciones de la ecuación. Al aplicar PUBS en este CRS se obtienen las UBFs [1] $while^+(s_0) = s_0$, que es asintóticamente precisa, y $for^+(s_0, m_0) = s_0 * m_0$, que no lo es. ■

Para resolver el problema de imprecisión en este ejemplo, debemos averiguar qué componente de COSTA la introduce. Hay cuatro posibilidades: (1) la compilación abstracta del RBR en el ACR; (2) el análisis de valor que infiere una postcondición en la ACR; (3) la transformación del ACR en el CRS; y (4) la resolución del CRS en la UBF. Tres de estas posibilidades se pueden descartar:

- El ACR de la Figura 12.2 es una abstracción precisa del programa JAVA de la Figura 12.1, en tanto que toda traza de la ACR trace corresponde a una ejecución de `main`. Por tanto se descarta la compilación abstracta.
- La postcondición $s_0 \geq s_1 \geq 0$ que el análisis de valor infiere para el procedimiento *while*, es la relación de entrada-salida más precisa para el bucle `while`. Esto descarta el análisis de valor; y
- Al examinar el CRS de la Figura 12.3, toda llamada a $for(s_0, m_0)$ admite un árbol de evaluación como el de la Figura 12.4, cuyo coste total es $s_0 * m_0$. Por tanto $for^+(s_0, m_0) = s_0 * m_0$ es una UBF precisa para este CRS, por lo que también se descarta el proceso de resolución de PUBS.

Solo queda el componente que transforma el ACR en el CRs, luego éste es el que

[1] En este capítulo, por concisión las UBFs se escriben sin el operador `nat`, pues en todos los ejemplos las expresiones siempre son no negativas.

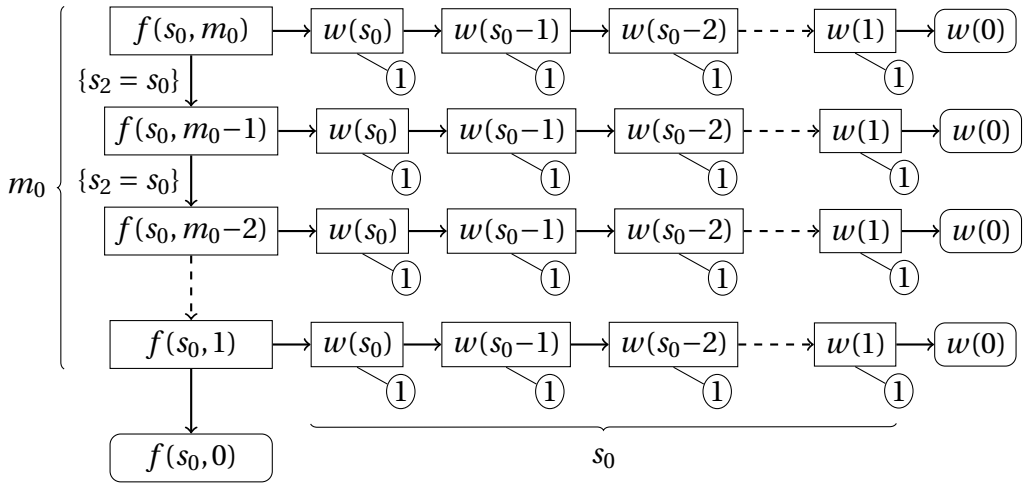


Figura 12.4: Árbol de evaluación para $for(s_0, m_0)$. Se abrevia *for*, *while* con f, w.

introduce la imprecisión. De hecho, el CRS contiene evaluaciones espúreas que no corresponden a ninguna traza de la ACR, como se muestra en este ejemplo.

Ejemplo 12.2. Considere el CRS de la Figura 12.3 y el árbol de evaluación de la Figura 12.4 para una llamada a $for(s_0, m_0)$. El árbol de evaluación incluye m_0 nodos que corresponden a las aplicaciones de la Ecuación (b), que forman la cadena vertical a la izquierda. De cada nodo salen dos aristas, que corresponden a las llamadas a *while* y a *for*. Como se indica en las aristas, en la llamada a *for* se escoge un valor s_2 para el primer parámetro tal que $s_2 = s_0$, lo cual satisface las restricciones de la Ecuación (b). Cada llamada a $while(s_0)$ da lugar a un subárbol (cadena horizontal) con s_0 nodos, que corresponden a la aplicación de la Ecuación (d), y cada uno tiene un coste local de 1. Por tanto, el coste de cada subárbol es s_0 y, dado que hay m_0 subárboles, el coste total es $s_0 * m_0$.

Veamos cómo este árbol de evaluación, y con él el coste de $s_0 * m_0$, no corresponde a ninguna traza del ACR. Para ello solo hay que considerar la primera aplicación de la Ecuación (b), que corresponde a la cadena horizontal más arriba en la Figura 12.4. Antes de ejecutar la llamada recursiva a $for(s_2, m_0 - 1)$, la llamada a $while(s_0)$ añade s_0 unidades al coste, y en la llamada recursiva se escoge un valor para s_2 tal que $s_2 = s_0$. Veamos por qué esto no sucede en la ACR. La Figura 12.5 muestra tres trazas de la ACR que corresponden a la primera iteración de $for(\langle s_0, m_0\rangle, \langle\rangle)$, y que difieren en cómo se ejecuta la llamada a $while(\langle s_0\rangle, \langle s_2\rangle)$, la cual es no determinista.

- La Traza (a) corresponde al caso en que *while* da s_0 iteraciones, en el cual el coste es s_0 y a la salida $s_2 = 0$.

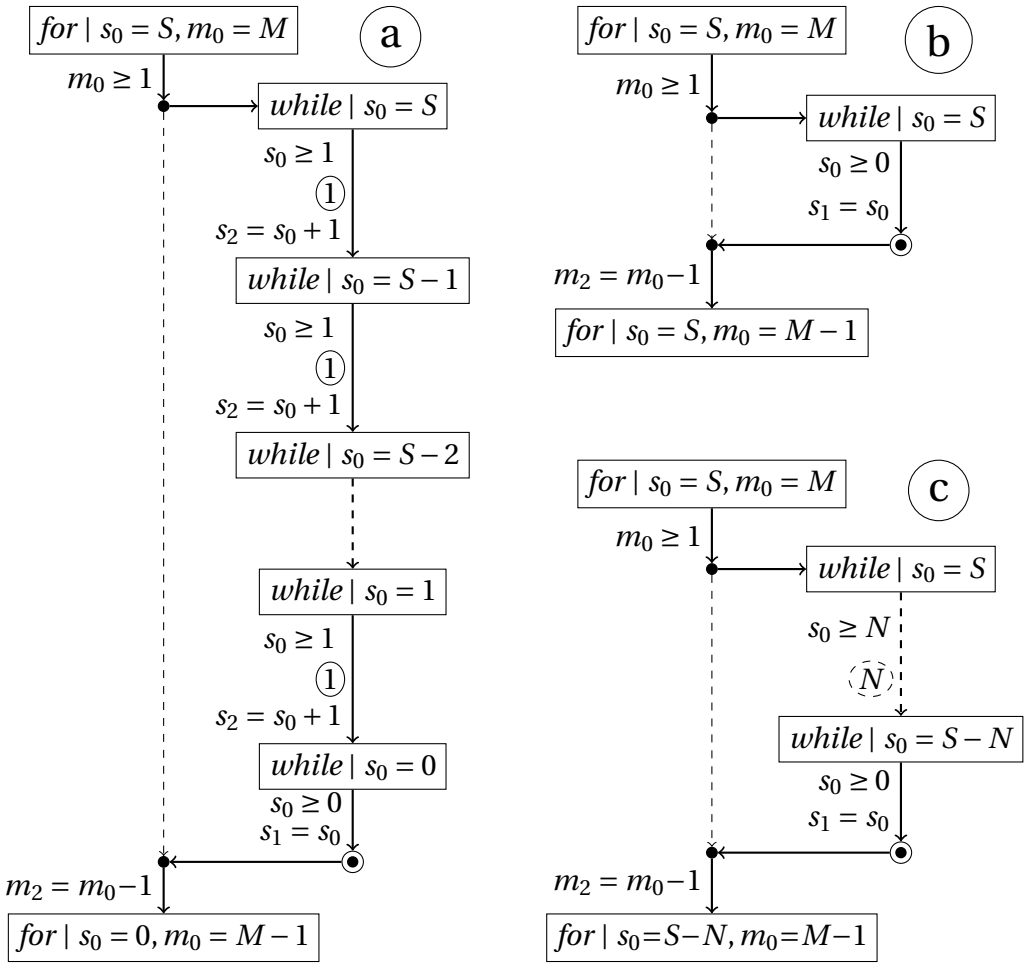


Figura 12.5: Fragmentos de traza de la ACR para una llamada a $for(\langle s_0\rangle, \langle m_0\rangle)$.

- La Traza (b) corresponde al caso en que *while* da 0 iteraciones, luego su coste es 0 y a la salida $s_2 = s_0$.
- La Traza (c) corresponde al caso en que *while* da N iteraciones, luego el coste es N y a la salida $s_2 = s_0 - N$.

El árbol de evaluación de la Figura 12.4 es espúreo porque, según las trazas ACR arriba, no sucede que el coste de $while(\langle s_0\rangle, \langle s_2\rangle)$ sea s_0 y que, a la vez, a la salida se tenga que $s_2 = s_0$. Este comportamiento mezcla el de las trazas (a) y (b). ■

Este ejemplo revela una dependencia entre la salida y el coste del procedimiento ACR *while*. Ésta es crucial para inferir una UBF precisa para el procedimiento *for*, pero en COSTA se pierde al transformar la Regla (b) de la ACR en la Ecuación (b), porque se quitan los parámetros de salida. En realidad, la CR de *while* representa con total precisión qué relación hay entre la entrada

y el coste de este procedimiento; asimismo, la postcondición representa con total precisión qué relación hay entre la entrada y la salida; pero ninguna de éstas representa la relación entre salida y coste. Para poder representar esta codependencia, habría que usar una forma de UBF que expresa el coste de un procedimiento como función de sus entradas y salidas.

Ejemplo 12.3. Para el procedimiento $while(\langle s_0 \rangle, \langle s_1 \rangle)$, la UBF más precisa definida en términos solo de su entrada s_0 es $while^+(s_0) = s_0$. Sin embargo, si se consideran las funciones que también dependen de su salida s_1, entonces se puede usar la UBF $while^+(s_0, s_1) = s_0 - s_1$ que da el coste exacto de *while*. ■

La idea de definir una UBF como función de la salida parece ir contra nuestra intuición. Una UBF se usa para estimar *estáticamente* el coste, esto es cuántos recursos se necesitan para ejecutarlo, pero el que la UBF dependa de esos parámetros parece requerir que se ejecute el programa. Ése no es en realidad nuestro caso. Al inferir UBFs, se distingue el procedimiento inicial (como *for*) y el resto (como *while*): para ése se puede definir una UBF precisa solo en términos de las entradas, pero para ello quizás haya que inferir para otros procedimientos unas UBFs que sí dependan de sus salidas. Por otra parte, luego se introducen las UBFs en el coste pico, que solo dependen de la entrada, pero que para inferirlas hay que usar las UBFs en el coste neto.

Para poder inferir UBFs que dependan en los parámetros de salida se necesita una nueva forma de CRSs que incluya esos parámetros. Puesto que los CRSs son de naturaleza declarativa, una solución es añadir las salidas de cada procedimiento ACR como entradas de la CR. Sin embargo, PUBS aun así falla en inferir la UBF para nuestro ejemplo, porque su método de resolución consiste en multiplicar el número de iteraciones de *for* (que es m_0) por el coste máximo de cada una (que es s_0). Por tanto, no basta añadir las parámetros de salida a los CRSs, sino que también se necesita un formas de resolución que tomen en cuenta la relación entre el coste de las varias iteraciones de *for*.

En las siguientes secciones se describen técnicas para calcular UBFs, para distintas nociones de coste, con las que se resuelve el problema de imprecisión. En vez de usar una nueva forma de CRSs, nuestras técnicas operan directamente sobre la ACR, que en cierto modo es un CRS con entradas y salidas.

2 Inferencia de UBFs en el Coste Neto

Usando la notación de la Sección 9.3, una traza completa y finita t para una llamada a $p(\bar{v}, \bar{y})$ empieza en el estado $\langle \bar{x} = \bar{v}, p(\bar{x}, \bar{y}) \rangle$ y acaba en $\langle \psi, \epsilon \rangle$, siendo ϵ una lista vacía de instrucciones. El coste neto de t es la suma del coste de

$$
\begin{array}{lll}
\widetilde{\phi}_a \equiv \forall s_0, m_0: & m_0{=}0 \wedge s_0 \geq 0 & \rightarrow \widetilde{f}(s_0, m_0) \geq 0 \\
\widetilde{\phi}_b \equiv \forall s_0, m_0, s_2, m_2: & \left(\begin{array}{l} m_0{\geq}1 \wedge s_0 \geq 0 \\ \wedge\, m_2 = m_0{-}1 \end{array}\right) & \rightarrow \widetilde{f}(s_0, m_0) \geq \widetilde{w}(s_0, s_2) + \widetilde{f}(s_2, m_2) \\
\widetilde{\phi}_c \equiv \forall s_0, s_1: & s_0 \geq 0 \wedge s_1 = s_0 & \rightarrow \widetilde{w}(s_0, s_1) \geq 0 \\
\widetilde{\phi}_d \equiv \forall s_0, s_1, s_2: & s_0 \geq 1 \wedge s_2 = s_0 - 1 & \rightarrow \widetilde{w}(s_0, s_1) \geq 1 + \widetilde{w}(s_2, s_1)
\end{array}
$$

Figura 12.6: VC del Coste Neto para el programa ACR de la Figura 12.2

todas las transiciones en la traza, lo que se denota como $acrcost(t)$ en la Sección 9.3. Fíjese que las UBFs en el coste neto se definen sobre las salidas del procedimiento, y por eso esta noción solo se aplica a trazas completas y finitas.

Para un programa que no usa release(e), el coste neto representa la cantidad de recursos necesarios para ejecutar el programa con seguridad. Sin embargo, si los programas usan release(e), entonces el coste neto no es suficiente, pues éste no sirve para estimar la cantidad de recursos necesarios para ejecutar con seguridad un programa. Por ejemplo, el coste neto de un programa que devuelve todos los recursos que adquiere es 0. Esto quedará claro en la Sección 12.3, pero por ahora suponemos que los programas no usan release(e).

Nuestro método para inferir UBFs en el coste neto se basa en las técnicas de análisis lógico de programas: en esencia, vemos las UBFs como una especificación y aplicamos el método de aserciones inductivas [34, 81] en dos pasos:

1. *Verificación*: A partir de un programa ACR y de un conjunto de UBFs candidatas para el coste neto, se muestra un procedimiento que verifica si esas UBFs son válidas. Para ello, del ACR se deriva una **condición de verificación** (*verification condition* o VC), que es una fórmula en la lógica de primer orden (*First Order Logic* o FOL), cuya validez implica que las UBFs candidatas son válidas.
2. *Inferencia*: Este procedimiento se convierte en uno de inferencia, para sintetizar UBFs en el coste neto, usando **plantillas** y QE.

Veamos ahora estos dos pasos, aplicados al ejemplo de la Sección 12.1. Primero veamos cómo se genera la VC.

Ejemplo 12.4. Para el programa ACR de la Figura 12.2, se deriva la VC $\widetilde{\Phi}$, que es la conjunción $\widetilde{\Phi} = \widetilde{\phi}_a \wedge \widetilde{\phi}_b \wedge \widetilde{\phi}_c \wedge \widetilde{\phi}_d$ de las cláusulas $\widetilde{\phi}_a, \ldots, \widetilde{\phi}_d$ de la Figura 12.6. Las funciones $\widetilde{w}(s_0, s_1)$ y $\widetilde{f}(s_0, m_0)$ son las UBFs candidatos en el coste neto de *while* y *for*, respectivamente. Como queda dicho, $\widetilde{w}(s_0, s_1)$ usa los parámetros tanto de entrada como de salida del procedimiento ACR *while*.

Cada cláusula $\widetilde{\phi}_i$ se genera de la regla ACR con la misma etiqueta en la Figura 12.2, y es de la forma $\forall \bar{x} : \varphi \rightarrow f \geq g$. La desigualdad $f \geq g$ indica que f, el coste neto del procedimiento, es mayor que la suma g del coste neto de cada

instrucción en el cuerpo de la regla. En esta suma g, el coste neto de acquire(e) es e, el de release(e) es $-e$, el de una llamada a procedimiento es la UBF candidata del procedimiento, y el de una restricción es 0. La cláusula dice que en el contexto φ, que es la conjunción de todas las restricciones en la regla, se tiene que cumplir $f \geq g$. El cuantificador universal indica que la desigualdad se ha de cumplir para todos los valores de las variables de la ACR que satisfagan φ.

Veamos en más detalle la cláusula $\widetilde{\phi}_b$. Dice que, para que $\widetilde{f}(s_0, m_0)$ sea una UBF válida en el coste neto de *for*, debe ser mayor que el coste neto $\widetilde{w}(s_0, s_2)$ de la llamada a *while*, más el coste neto $\widetilde{f}(s_2, m_2)$ de la llamada recursiva a *for*; y ésto se ha de cumplir para cualquier valor de s_0, m_0 que satisfaga las restricciones $m_0 \geq 1 \wedge s_0 \geq 0 \wedge m_2 = m_0 - 1$ de la Regla (b). ■

Dada una definición concreta de las UBFs candidatas en el coste neto, se puede comprobar si son válidas simplemente sustituyéndolas en la VC y comprobando si ésta es válida, para lo que se puede usar un resolutor SMT.

Ejemplo 12.5. Consideremos $\widetilde{f}(s_0, m_0){=}s_0$ y $\widetilde{w}(s_0, s_1){=}s_0 - s_1$, las cuales son las UBFs más precisas para el coste neto de *for* y *while* que dependen de las entradas y salidas de cada uno. Al sustituir estas funciones en la VC de la Figura 12.6 se obtiene:

$$\begin{array}{lll}
\widetilde{\phi}_a \equiv \forall s_0, m_0: & m_0{=}0 \wedge s_0 \geq 0 & \rightarrow s_0 \geq 0 \\
\widetilde{\phi}_b \equiv \forall s_0, m_0, s_2, m_2: & m_0{\geq}1 \wedge s_0 \geq 0 \wedge m_2 = m_0{-}1 & \rightarrow s_0 \geq s_0 - s_2 + s_2 \\
\widetilde{\phi}_c \equiv \forall s_0, s_1: & s_0 \geq 0 \wedge s_1 = s_0 & \rightarrow s_0 - s_1 \geq 0 \\
\widetilde{\phi}_d \equiv \forall s_0, s_1, s_2: & s_0 \geq 1 \wedge s_2 = s_0 - 1 & \rightarrow s_0 - s_1 \geq 1 + s_2 - s_1
\end{array}$$

Dado que la fórmula formula $\widetilde{\Phi} = \widetilde{\phi}_a \wedge \widetilde{\phi}_b \wedge \widetilde{\phi}_c \wedge \widetilde{\phi}_d$ es válida, estas UBFs candidatas son válidas. Por otro lado, si en vez de $\widetilde{w}(s_0, s_1) = s_0 - s_1$ se usa $\widetilde{w}(s_0, s_1) = s_0$, que es la UBF más precisa que solo usa las entradas de *while*, se obtiene:

$$\begin{array}{lll}
\widetilde{\phi}_a \equiv \forall s_0, m_0: & m_0{=}0 \wedge s_0 \geq 0 & \rightarrow s_0 \geq 0 \\
\widetilde{\phi}_b \equiv \forall s_0, m_0, s_2, m_2: & m_0{\geq}1 \wedge s_0 \geq 0 \wedge m_2 = m_0{-}1 & \rightarrow s_0 \geq s_0 + s_2 \\
\widetilde{\phi}_c \equiv \forall s_0, s_1: & s_0 \geq 0 \wedge s_1 = s_0 & \rightarrow s_0 \geq 0 \\
\widetilde{\phi}_d \equiv \forall s_0, s_1, s_2: & s_0 \geq 1 \wedge s_2 = s_0 - 1 & \rightarrow s_0 \geq 1 + s_2
\end{array}$$

En este caso $\widetilde{\phi}_c$ y $\widetilde{\phi}_d$ son válidas, en tanto que $\widetilde{w}$ es una UBF válida para el coste neto de *while*. Sin embargo, $\widetilde{\phi}_b$ no es válida pese a que $\widetilde{f}(s_0, m_0) = s_0$ es una UBF válida para *for*. La razón es que no se puede verificar que $\widetilde{f}$ es válida usando la UBF solo de entradas $\widetilde{w}(s_0, s_1) = s_0$ de *while*. ■

En vez de verificar que unas UBFs candidatas son válidas, nos interesa más sintetizar éstas directamente. Esto se puede formular como buscar unas UBFs

en el coste neto $\{\widetilde{f}_1, \ldots, \widetilde{f}_k\}$ para las que la VC sea válida, pero éste es un problema de la lógica de segundo orden que no es fácil de resolver. Un método para evitar este problema de segundo orden es usar funciones plantilla, las cuales restringen la forma de las funciones que se buscan [124, 127]. Una **plantilla** es una función con una estructura prefijada, definida sobre los parámetros del procedimiento, que también contiene otras variables llamadas parámetros de la plantilla. Al usar plantillas se reduce el problema de segundo orden a uno de FOL, que es buscar valores para los parámetros de la plantilla.

Ejemplo 12.6. Considere estas UBF plantilla para el coste neto de *for* y *while*:

$$\widetilde{f}(s_0, m_0) = f_\mathrm{s} s_0 + f_\mathrm{m} m_0 + f_\mathrm{c} \qquad \widetilde{w}(s_0, s_1) = w_\mathrm{s} s_0 + w_\mathrm{t} s_1 + w_\mathrm{c}$$

Las variables $\{f_\mathrm{s}, f_\mathrm{m}, f_\mathrm{c}, w_\mathrm{s}, w_\mathrm{t}, w_\mathrm{c}\}$ son los parámetros de la plantilla, mientras que s_0, m_0, s_1 son las variables de la ACR. Al sustituir estas plantillas en la VC de la Figura 12.6 se obtiene las siguientes fórmulas FOL:

$$\begin{array}{lll}
\widetilde{\phi}_a \equiv \forall s_0, m_0\colon & m_0{=}0 \wedge s_0 \geq 0 & \rightarrow f_\mathrm{s} s_0 + f_\mathrm{m} m_0 + f_\mathrm{c} \geq 0 \\
\widetilde{\phi}_b \equiv \forall \left\{ \begin{smallmatrix} s_0, m_0, \\ s_2, m_2 \end{smallmatrix} \right\}\colon & \left(\begin{smallmatrix} m_0 {\geq} 1 \wedge s_0 \geq 0 \\ \wedge m_2 = m_0 {-} 1 \end{smallmatrix} \right) & \rightarrow f_\mathrm{s} s_0 + f_\mathrm{m} m_0 \geq \left(\begin{smallmatrix} w_\mathrm{s} s_0 + w_\mathrm{t} s_2 + w_\mathrm{c} \\ + f_\mathrm{s} s_2 + f_\mathrm{m} m_2 \end{smallmatrix} \right) \\
\widetilde{\phi}_c \equiv \forall s_0, s_1\colon & s_0 \geq 0 \wedge s_1 = s_0 & \rightarrow w_\mathrm{s} s_0 + w_\mathrm{t} s_1 + w_\mathrm{c} \geq 0 \\
\widetilde{\phi}_d \equiv \forall s_0, s_1, s_2\colon & s_0 \geq 1 \wedge s_2 = s_0 - 1 & \rightarrow w_\mathrm{s} s_0 + w_\mathrm{t} s_1 + w_\mathrm{c} \geq 1 + w_\mathrm{s} s_2 + w_\mathrm{t} s_1 + w_\mathrm{c}
\end{array}$$

En éstas fórmulas, las variables de la ACR están cuantificadas universalmente, pero los parámetros de la plantilla son variables libres.

Resolver $\widetilde{\Phi}$ significa encontrar valores de $\{f_\mathrm{s}, f_\mathrm{m}, f_\mathrm{c}, w_\mathrm{s}, w_\mathrm{t}, w_\mathrm{c}\}$ para los que la fórmula sea válida. Dichos valores definen instancias de las plantillas que son UBFs válidas. Para resolver $\widetilde{\Phi}$, primero se aplica un procedimiento QE, que suprime las variables cuantificadas (las de la ACR), y deja una fórmula equivalente sin cuantificadores sobre los parámetros de la plantilla, llamada $\widetilde{\Psi} = \widetilde{\psi}_a \wedge \widetilde{\psi}_b \wedge \widetilde{\psi}_c \wedge \widetilde{\psi}_d$, siendo:

$$\begin{array}{ll}
\widetilde{\psi}_a \equiv f_\mathrm{s} \geq 0 \wedge f_\mathrm{c} \geq 0 & \widetilde{\psi}_c \equiv\ w_\mathrm{s} + w_\mathrm{t} \geq 0 \wedge w_\mathrm{c} \geq 0 \\
\widetilde{\psi}_b \equiv f_\mathrm{s} \geq w_\mathrm{s} \wedge f_\mathrm{m} \geq w_\mathrm{c} \wedge f_\mathrm{s} + w_\mathrm{t} = 0 & \widetilde{\psi}_d \equiv w_\mathrm{s} \geq 1
\end{array}$$

Como $\widetilde{\Phi}$ y $\widetilde{\Psi}$ son equivalentes, cualquier solución de $\widetilde{\Psi}$ también lo es de $\widetilde{\Phi}$. Por ejemplo, $\{f_\mathrm{s} = 1, f_\mathrm{m} = 0, f_\mathrm{c} = 0, w_\mathrm{s} = 1, w_\mathrm{t} = -1, w_\mathrm{c} = 0\}$ es una solución de $\widetilde{\Psi}$, que da las UBFs en el coste neto UBFs $\widetilde{f}(s_0, m_0) = s_0$ y $\widetilde{w}(s_0, s_1) = s_0 - s_1$. ■

Inferencia de Cotas Inferiores. Es fácil de adaptar nuestro método para calcular LBFs en el coste neto, sencillamente cambiando cada ≥ por ≤ en la VC de la Figura 12.6.

```
1   //@requires n>=1
2   void p(int n) {
3     if (n > 1) {
4       int m = q(n);
5       // gc A
6       p(n-m);
7       // gc B
8     }
9   }
```

```
10  //@requires n>=2
11  int q(int n) {
12    int i = n/2;
13    do {
14      A x = new A();
15      B y = new B();
16      i--;
17    } while(i>0 && coin());
18    return n/2 - i;
19  }
```

Figura 12.7: Código JAVA para el ejemplo de espacio pico en memoria.

Coste Neto y no terminación. Como el coste neto se define para trazas completas y finitas, nuestro método no es adecuado para acotar el coste de un programa no terminante. Esto se resuelve al final de la próxima sección.

3 Inferencia de UBFs en el Coste Pico

El consumo de un recurso que solo se adquiere, como el número de instrucciones, se analiza con un modelo de coste **acumulativo**, esto es, uno que solo introduce instrucciones acquire(e) en el ACR. Sin embargo, hay recursos que se adquieren y se devuelven durante la ejecución, como por ejemplo la memoria en el montículo en presencia de un recolector de basura (*garbage collector* o GC) [13]. El consumo de tales recursos se analiza con un modelo de coste **no acumulativo**, que introduce en el programa ACR program instrucciones para adquirir y para devolver recursos. Para permitir tales modelos de coste, añadimos al ACR una instrucción release(e), que devuelve e unidades de recursos, cuya semántica viene dada por esta regla:

$$\frac{eval(e,\psi) = v \geq 0}{\langle \psi, \mathsf{release}(e) \cdot \bar{a} \rangle \overset{-v}{\rightsquigarrow} \langle \psi, \bar{a} \rangle}$$

la cual anota la transición con un valor negativo $-v$. Veamos un ejemplo del uso de estas instrucciones ACR.

Ejemplo 12.7. Considere el programa JAVA en la Figura 12.7. El método `q` recibe un entero `n`, ejecuta un bucle entre 1 y `n/2` iteraciones, y devuelve el número de iteraciones realizadas. En cada iteración se crea un objeto de la clase `A` y uno de la clase `B`. El método recursivo `p` implementa un bucle que en cada iteración llama a `q` con el valor de `n`, y luego realiza una llamada recursiva con el

(a) $p(\langle n_0\rangle, \langle\rangle) \leftarrow$
$\quad n_0 = 1.$
(b) $p(\langle n_0\rangle, \langle\rangle) \leftarrow$
$\quad n_0 \geq 2,$
$\quad q(\langle n_0\rangle, \langle m_2\rangle),$
$\quad n_2 = n_0 - m_2,$
$\quad p(\langle n_2\rangle, \langle\rangle),$
$\quad \mathsf{release}(m_2).$

(c) $q(\langle n_0\rangle, \langle m_1\rangle) \leftarrow$
$\quad n_0 \geq 2,$
$\quad i_2 = n_0/2 - 1,$
$\quad \mathsf{acquire}(2),$
$\quad l(\langle i_2\rangle, \langle i_3\rangle),$
$\quad m_1 = n_0/2 - i_3,$
$\quad \mathsf{release}(m_1).$

(d) $l(\langle i_0\rangle, \langle i_1\rangle) \leftarrow$
$\quad i_0 \geq 0,$
$\quad i_1 = i_0.$
(e) $l(\langle i_0\rangle, \langle i_1\rangle) \leftarrow$
$\quad i_0 \geq 1,$
$\quad \mathsf{acquire}(2),$
$\quad i_2 = i_0 - 1,$
$\quad l(\langle i_2\rangle, \langle i_1\rangle).$

Figura 12.8: Programa ACR para el programa JAVA de la Figura 12.7

contador reducido en `m` (el número de iteraciones que `q` realiza). Se quiere analizar cuánto espacio en memoria (medido en objetos) hace falta para ejecutar el programa, suponiendo que el GC recoge todas las instancias de la clase `A` tras la llamada a `q` (Línea 5), y todas las de la clase `B` tras la llamada recursiva (Línea 7). El programa ACR se muestra en la Figura 12.8. Se usa $\mathsf{release}(m_2)$ para representar el efecto de aplicar el GC en ese punto del programa. ■

Cuando se analiza un programa con un modelo de coste acumulativo, basta con un UBF en el coste neto para estimar cuántos recursos hacen falta para ejecutar el programa de manera segura. En cambio, si éste se analiza con un modelo de coste no acumulativo, entonces se necesita estimar el **coste pico**, que es la cantidad de recursos retenidos (adquidos pero no devueltos) en cualquier punto de la ejecución. El coste pico de una traza t se define como máx$\{\, acrcost(t') \mid t'$ es un prefijo de $t\}$. Una función p^+ es un UBF para el coste pico del procedimiento p si, para cualquier traza t que empieza en $\langle \bar{x} = \bar{v}, p(\bar{x}, \bar{y})\rangle$, y cualquier prefijo t' de t, se tiene que $p^+(\bar{v}) \geq acrcost(t')$. Fíjese que esta definición se refiere a trazas completas, finitas o no.

Ejemplo 12.8. Cada ejecución de `p(n)` crea $2n$ objetos, pero todos ellos se liberan antes de que acabe la ejecución, por lo que el coste neto de este programa 0 (esto es lo que el análisis de la Sección 12.2 infiere). Sin embargo, debido al GC, el programa solo puede retener hasta n objetos: éste es el coste pico. Al analizar este programa con COSTA usando un modelo de coste que cuenta el número de objetos [13], se obtiene una AUBF imprecisa en $O(n^2)$. ■

Para manejar este ejemplo, se complementa la UBF en el coste neto con una UBF en el coste pico. Para un procedimiento ACR, una UBF en el coste pico es una función sobre los parámetros de entrada que, para cualquier valor de las entradas, cualquier traza que empiece con esa entrada tiene un coste pico menor que la UBF. Fíjese que las UBFs en el coste pico se definen solo en función de las entradas. Ello es así porque cubren trazas parciales, y así pueden

$$
\begin{array}{lll}
\widehat{\phi}_a \equiv \ \forall \bar{w}_a : n_0 = 1 & \rightarrow & \widehat{p}(n_0) \geq 0 \\
\widehat{\phi}_b \equiv \ \forall \bar{w}_b : (n_0 \geq 2 & \rightarrow & \widehat{p}(n_0) \geq 0 \quad) \\
\qquad \wedge (n_0 \geq 2 & \rightarrow & \widehat{p}(n_0) \geq \widehat{q}(n_0) \quad) \\
\qquad \wedge (n_0 \geq 2 \wedge n_2 = n_0 - m_2 & \rightarrow & \widehat{p}(n_0) \geq \widetilde{q}(n_0, m_2) + \widehat{p}(n_2) \quad) \\
\widehat{\phi}_c \equiv \ \forall \bar{w}_c : (n_0 \geq 2 \wedge n_0 = 2i_2 + 2 & \rightarrow & \widehat{q}(n_0) \geq 2 \quad) \\
\qquad \wedge (n_0 \geq 2 \wedge n_0 = 2i_2 + 2 & \rightarrow & \widehat{q}(n_0) \geq 2 + \widehat{l}(i_2) \quad) \\
\widehat{\phi}_d \equiv \ \forall \bar{w}_d : i_0 \geq 0 & \rightarrow & \widehat{l}(i_0) \geq 0 \\
\widehat{\phi}_e \equiv \ \forall \bar{w}_e : (i_0 \geq 1 & \rightarrow & \widehat{l}(i_0) \geq 2 \quad) \\
\qquad \wedge (i_0 \geq 1 \wedge i_2 = i_0 - 1 & \rightarrow & \widehat{l}(i_0) \geq 2 + \widehat{l}(i_2) \quad)
\end{array}
$$

$$
\begin{array}{lll}
\widetilde{\phi}_a \equiv \ \forall \bar{w}_a : n_0 = 1 & \rightarrow & \widetilde{p}(n_0) \geq 0 \\
\widetilde{\phi}_b \equiv \ \forall \bar{w}_b : n_0 \geq 2 \wedge n_2 = n_0 - m_2 & \rightarrow & \widetilde{p}(n_0) \geq \widetilde{q}(n_0, m_2) + \widetilde{p}(n_2) - m_2 \\
\widetilde{\phi}_c \equiv \ \forall \bar{w}_c : \left(\begin{array}{l} n_0 \geq 2 \wedge n_0 = 2i_2 + 2 \\ 2m_1 = n_0 - 2i_3 \end{array} \right) & \rightarrow & \widetilde{q}(n_0, m_1) \geq 2 + \widetilde{l}(i_2, i_3) - m_1 \\
\widetilde{\phi}_d \equiv \ \forall \bar{w}_d : i_0 \geq 0 \wedge i_1 = i_0 & \rightarrow & \widetilde{l}(i_0, i_1) \geq 0 \\
\widetilde{\phi}_e \equiv \ \forall \bar{w}_e : i_0 \geq 1 \wedge i_2 = i_0 - 1 & \rightarrow & \widetilde{l}(i_0, i_1) \geq 2 + \widetilde{l}(i_2, i_1)
\end{array}
$$

Figura 12.9: VC del coste neto y coste pico del programa ACR de la Figura 12.8

acotar el coste de ejecuciones no terminantes, cuyo coste no puede expresarse en términos de las salidas. Sin embargo, para inferir las UBFs en el coste pico se necesitan las del coste neto, que si dependen de entradas y salidas.

Como en el coste neto, nuestro método se divide en dos pasos: del programa ACR y para algunas UBFs candidatas se deriva una VC, cuya validez implica que las UBFs candidatas son válidas; y luego se usan UBFs plantilla y QE para convertir este procedimiento en uno de inferencia. Empecemos por explicar la intuición para el coste pico. En una regla ACR $p(\bar{x}, \bar{y}) \leftarrow q_1(\bar{x}, \bar{w}), q_2(\bar{w}, \bar{y})$, al ejecutar p el coste pico puede se puede dar al ejecutar o bien q_1 o bien q_2. El primer caso implica que el coste pico de p debe ser mayor que el de q_1; y el segundo caso implica que el coste pico de p es mayor que el coste pico de q_2 más el coste neto de q_1. Además, el coste pico debe ser no negativo. A continuación se aplica esta intuición al ejemplo anterior.

Ejemplo 12.9. Del programa ACR de la Figura 12.8 se deriva la VC $\widehat{\Phi} = \widehat{\phi}_a \wedge \cdots \wedge \widehat{\phi}_e \wedge \widetilde{\phi}_a \wedge \cdots \wedge \widetilde{\phi}_e$, donde las $\widetilde{\phi}_i$ son cláusulas para el coste neto (igual que en la Sección 12.2) y las $\widehat{\phi}_i$ son las cláusulas del coste pico. Estas cláusulas se muestran en la Figura 12.9, donde $\bar{w}_i$ indica las variables de la regla ACR (i).

Cada cláusula $\widehat{\phi}_i$ se deriva de la regla ACR con el mismo nombre (i), siguiendo la intuición antes descrita. Por ejemplo, la cláusula $\widehat{\phi}_b$ incluye tres subcláusulas: la primera dice que el coste pico de p es no negativo; la segunda, que es mayor que el de q; y la tercera, que es mayor que el coste pico de la llamada re-

$$
\begin{array}{ll}
\widehat{\psi}_a \equiv & \widehat{p}_n + \widehat{p}_c \geq 0 \\
\widehat{\psi}_b \equiv & (\widehat{p}_n \geq 0 \wedge 2\widehat{p}_n + \widehat{p}_c \geq 0) \ \wedge \ (\widehat{p}_n \geq \widehat{q}_n \wedge 2\widehat{p}_n + \widehat{p}_c \geq 2\widehat{q}_n + \widehat{q}_c) \\
& \quad \wedge \ (\widetilde{q}_n \leq 0 \wedge 2\widetilde{q}_n + \widetilde{q}_c \leq 0 \wedge \widetilde{p}_n = \widetilde{q}_m) \\
\widehat{\psi}_c \equiv & (\widehat{q}_n \geq 0 \wedge 2\widehat{q}_n + \widehat{q}_c \geq 2) \ \wedge \ (2\widehat{q}_n \geq \widehat{l}_i \wedge 2\widehat{q}_n + \widehat{q}_c \geq \widehat{l}_c + 2) \\
\widehat{\psi}_d \equiv & \widehat{l}_i \geq 0 \wedge \widehat{l}_c \geq 0 \\
\widehat{\psi}_e \equiv & (\widehat{l}_i \geq 0 \wedge \widehat{l}_i + \widehat{l}_c \geq 2) \ \wedge \ (\widehat{l}_i \geq 2)
\end{array}
$$

$$
\begin{array}{ll}
\widetilde{\psi}_a \equiv & \widetilde{p}_c + \widetilde{p}_n \geq 0 \\
\widetilde{\psi}_b \equiv & \widetilde{p}_n = \widetilde{q}_m - 1 \wedge \widetilde{q}_n \leq 0 \wedge 2\widetilde{q}_n + \widetilde{q}_c \leq 0 \\
\widetilde{\psi}_c \equiv & 2\widetilde{q}_n \geq \widetilde{l}_i + \widetilde{l}_j \wedge \widetilde{q}_m + \widetilde{l}_j = -1 \wedge 2\widetilde{q}_n + \widetilde{q}_c \geq \widetilde{l}_j + \widetilde{l}_c + 2 \\
\widetilde{\psi}_d \equiv & \widetilde{l}_i + \widetilde{l}_j \geq 0 \wedge \widetilde{l}_c \geq 0 \\
\widetilde{\psi}_e \equiv & \widetilde{l}_i \geq 2
\end{array}
$$

Figura 12.10: Cláusulas sin cuantificadores que se obtienen al poner en la VC de la Figura 12.9 las UBFs plantilla del Ejemplo 12.10, y aplicar QE.

cursiva p más el coste neto de las instrucciones anteriores, esto es, que el coste neto $\widetilde{q}(n_0, m_2)$ de la llamada a q. ■

Para verificar si unas UBFs candidatas son válidas, solo hay que sustituirlas en la VC y comprobar la validez de la fórmula. Aquí también interesa más inferir que verificar UBFs, y para ello se pueden usar UBFs plantilla y QE.

Ejemplo 12.10. Considere las siguientes UBF plantilla para el programa ACR de la Figura 12.8:

$$
\begin{array}{lll}
\widehat{p}(n_0) = \widehat{p}_n n_0 + \widehat{p}_c & \widehat{q}(n_0) = \widehat{q}_n n_0 + \widehat{q}_c & \widetilde{q}(n_0, m_1) = \widetilde{q}_n n_0 + \widetilde{q}_m m_1 + \widetilde{q}_c \\
\widetilde{p}(n_0) = \widetilde{p}_n n_0 + \widetilde{p}_c & \widehat{l}(i_0) = \widehat{l}_i i_0 + \widehat{l}_c & \widetilde{l}(i_0, i_1) = \widetilde{l}_i i_0 + \widetilde{l}_j i_1 + \widetilde{l}_c
\end{array}
$$

Al sustituirlas en la VC de la Figure 12.9, y luego aplicar la QE, se obtiene las fórmulas sin cuantificadores $\widehat{\Psi} = \widehat{\psi}_a \wedge \cdots \wedge \widehat{\psi}_e$ y $\widetilde{\Psi} = \widetilde{\psi}_a \wedge \cdots \wedge \widetilde{\psi}_e$ donde $\widetilde{\psi}_i$ y $\widehat{\psi}_i$ se muestran en la Figura 12.10. Cada cláusula $\widehat{\psi}_i$ (o $\widetilde{\psi}_i$) se obtiene al aplicar la QE a la cláusula $\widehat{\phi}_i$ (o $\widetilde{\phi}_i$). Cualquier solución de $\widehat{\Psi} \wedge \widetilde{\Psi}$ define una instancia de las UBFs plantilla que son UBFs válidas, tanto para el coste pico como para el coste neto. En particular, esta solución:

$$
\begin{array}{llllllll}
\widehat{p}_n = 1 & \widehat{p}_c = 0 & & \widehat{q}_n = 1 & \widehat{q}_c = 0 & & \widetilde{q}_n = 0 & \widetilde{q}_m = 1 \quad \widetilde{q}_c = 0 \\
\widetilde{p}_n = 0 & \widetilde{p}_c = 0 & & \widehat{l}_i = 2 & \widehat{l}_c = 0 & & \widetilde{l}_i = 2 & \widetilde{l}_j = -2 \quad \widetilde{l}_c = 0
\end{array}
$$

da la siguientes UBFs válidas para el coste neto y pico de los procedimientos del programa ACR de la Figura 12.8:

$$
\begin{array}{lll}
\widehat{p}(n_0) = n_0 & \widehat{q}(n_0) = n_0 & \widetilde{q}(n_0, m_1) = m_1 \\
\widetilde{p}(n_0) = 0 & \widehat{l}(i_0) = 2i_0 & \widetilde{l}(i_0, i_1) = 2i_0 - 2i_1
\end{array}
$$

Fíjese que la UBF en el coste neto de q solo depende del parámetro de salida m_1 de q. ■

Terminemos esta sección con un ejemplo de no terminación.

Ejemplo 12.11. Considere el siguiente programa ACR no terminante, definido por esta única regla:

$$p(\langle n_0\rangle, \langle\rangle) \leftarrow n_0 \geq m \geq 0, \mathsf{acquire}(m), n_2 = n_0 - m, p(\langle n_2\rangle, \langle\rangle).$$

El procedimiento p recibe un entero no negativo n_0, escoge de manera no determinista un valor no negativo $m \leq n_0$, añade m al coste, y llama recursivamente a p con $n_0 - m$. El coste pico de este programa es n_0, pues ninguna traza parcial tiene un coste mayor que n_0, y hay trazas parciales que tienen ese coste. Esta UBF en el coste pico UBF se puede inferir siguiendo el método descrito en esta sección. Para más detalles, véase el Ejemplo 9 del artículo. ■

4 Relación con el Análisis Amortizado

Veamos ahora una relación interesante entre las UBFs en el coste neto y las funciones de potencial usadas en el análisis amortizado automático de coste. Esto ofrece una explicación basada en la semántica de por qué el análisis amortizado obtiene UBFs más precisas.

Para un programa ACR, una función de potencial asigna a cada estado un número no negativo, el potencial de ese estado, el cual puede interpretarse como la cantidad de recursos disponibles en ese estado. Un análisis amortizado automático de coste [79, 73] calcularía para cada procedimiento ACR $p(\bar{x}, \bar{y})$ dos funciones de potencial: una para la entrada $P_p(\bar{x})$, y una para la salida $Q_p(\bar{y})$. De manera intuitiva, el potencial de entrada $P_p(\bar{x})$ debe ser suficiente para pagar el coste de ejecutar $p(\bar{x}, \bar{y})$, y, a la salida, dejar al menos $Q_p(\bar{y})$ unidades para pagar el coste del resto de la ejecución. Por tanto, si c es el coste neto de p, entonces se debe cumplir que $P_p(\bar{x}) \geq c + Q_p(\bar{y})$. Esto se puede reescribir como $P_p(\bar{x}) - Q_p(\bar{y}) \geq c$, luego la diferencia $P_p(\bar{x}) - Q_p(\bar{y})$ es una UBF para el coste neto de p, definido sobre las entradas y salidas de p.

Estas funciones de potencial son por tanto UBFs del tipo descrito en la Sección 12.2, pero solo de una forma especial. Hay casos en que la UBF no se puede expresar como una diferencia entre funciones, estando una definida solo sobre la entrada y la otra solo sobre la salida, sino que la UBF preciso para esos casos incluye términos que dependen de entradas y de salidas. Hay que señalar, además, que nuestra definición de UBF en el coste neto es aplicable a cualquier tipo de expresiones y funciones, en vez de estar restringidas a funciones de potencial lineales o polinómicas como [73].

5 Evaluación Experimental

Implementación. Hemos implementado nuestro método en un prototipo, llamado ACRP. El programa ocupa menos de 1500 líneas de código HASKELL. ACRP toma como entrada un archivo de texto que, en una sintaxis à-la PROLOG, contiene un programa ACR y un UBF plantilla para el coste neto y para el coste pico de cada procedimiento. ACRP genera la VC para el coste neto y el coste pico para ese programa ACR. Después, ACRP invoca los métodos de QE que proporciona REDLOG [56] (una extensión del sistema de álgebra por computador REDUCE [68]), para que calcule las restricciones sobre los parámetros de las UBFs plantilla. ACRP usa la teoría de *campos reales cerrados*, que nos permite usar un amplio rango de UBFs plantilla, incluyendo polinomios multivariable, así como las operaciones máx y mín. ACRP proporciona una opción para usar la combinación de REDLOG, QEPCAD y SLFQ descrita [127], lo cual simplifica las restricciones. ACRP devuelve las restricciones como un *script* en SMTLIB2 [29], usando la lógica de *aritmética real no lineal* (`QF_NRA`). Este *script* se puede dar a un resolutor SMT como Z3 [53], para obtener una solución de los parámetros.

Experimentos de Precisión. Hemos escrito varios programas ACR que corresponden a ejemplos de la literatura. Algunos de estos ejemplos son los métodos para operar en un contador binario [45, §17], o algunos de los bucles anidados que se analizan en los artículos del proyecto SPEED [65, 66, 64], Para estos ejemplos se obtienen las UBFs precisas. En muchos de ellos, había al menos un procedimiento para el que hacía falta usar una UBF para el coste neto que incluyera variables de salida. Esto indica la presencia en esos programas de la codependencia entre salida y coste, así como la eficacia de nuestro enfoque al análisis de coste basado en el uso de aserciones inductivas. Sin embargo, también encontramos que algunos programas a menudo descritos como teniendo un coste amortizado, como las operaciones en un array dinámico [45, §17] o en una cola construida con dos pilas [103], podían resolverse *sin* usar UBFs que dependieran en los parámetros de salida. Esto nos indicó que la codependencia entre salida y coste no era importante en esos programas, y que en principio sería posible resolver éstos a una AUBF precisa solo con usar un resolutor de CRs más preciso. Ésta fue la inspiración para la contribución que se describe en el Capítulo 13.

Límites de Escalabilidad. Debido al gran coste de los procedimientos de QE para la teoría de los campos reales, nuestro procedimiento solo se pudo aplicar a ejemplos pequeños, y no escala para programas grandes. Ello restringe la aplicación práctica de nuestro enfoque, pero no su importancia teórica. Por otra parte, nuestro enfoque se puede aplicar con cualquier conjunto de UBFs

plantilla, siempre que estén soportados por el procedimiento de QE que se use. En este sentido, la elección del procedimiento de QE y de la teoría FOL determina el ámbito y la escalabilidad de la técnica. En concreto, para evitar estas limitaciones en el Capítulo 13 nos restringimos a los procedimientos de QE para la aritmética real lineal, los cuales son más escalables.

6 Trabajo Relacionado

Análisis Amortizado Automático. Este método se introdujo en la tesis doctoral de Jost [79], y luego se ha extendido en varias direcciones [35, 80, 122, 76, 73].

Método de Aserciones Inductivas. La idea básica de nuestro método, ver las UBFs como especificaciones del programa, y usar el método de aserciones inductivas para verificarlas y sintetizarlas, ya había sido usada por Wegbreit [134] para estimar el coste promedio. La QE para los reales también se ha usado antes en el análisis de coste, para inferir UBFs [19], o para verificarlos [52].

Bucles Sencillos. Las técnicas de análisis de coste presentadas en el proyecto SPEED [64, 66, 140, 65], pueden tratar una forma particular de la codependencia entre salida y coste, que ocurre en algunos bucles anidados, en los que la variable del control del bucle externo se modifica en el bucle interno. A diferencia del nuestro, su enfoque no puede manejar la recursión múltiple.

7 Contenido Adicional

En este capítulo se han descrito las contribuciones de [18], sobre los límites del enfoque clásico de análisis de coste: I) se explica que estas limitaciones surgen por ignorar las salidas de los procedimientos ACR, y se ve que esta codependencia es esencial para inferir UBFs precisas; II) para superar estas limitaciones, se introducen los UBFs en el coste neto y en el coste pico, y se ve una técnica para inferirlos; y III) se explica qué relación hay con el análisis automático amortizado de coste, y por qué el análisis amortizado puede ser más preciso que el enfoque clásico.

El artículo presenta de manera formal la sintaxis y semántica de la ACR, las nociones de coste neto y coste pico de un programa ACR, la derivación a partir de un programa ACR de las VCs para el coste neto y pico, y los teoremas de corrección junto con sus demostraciones.

13 | Resolución Lógica de CRS

En este capítulo se describe un método preciso y escalable para resolver CRSs en UBFs, que se sitúa a medio camino entre el método escalable de PUBS y el método preciso del Capítulo 12. En concreto, se propone una técnica para resolver CRs aisladas, que primero descompone la CR en varias CRs **atómicas**, usa razonamiento local preciso para inferir una UBF para cada una, y las combina en una UBF de la CR original. Para el razonamiento local, se define la UBF como la solución de una fórmula en FOL, como se hacía en el Capítulo 12. Esta contribución se presentó en este artículo:

> DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Precise Cost Analysis via Local Reasoning. In Dang Van Hung and Mizuhito Ogawa, editors of the Proceedings of *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 8172 of *Lecture Notes in Computer Science*, pages 319–333. Springer, October 2013

que se puede encontrar en la Página 253.

Descripción

Como se vio en la Sección 9.5, en el método de PUBS [5] para resolver una CR aislada se multiplica el coste máximo de cada ecuación por el número máximo de veces que ésta puede ser aplicada. Este método es eficiente y trata una amplia clase de CRs, pero a veces da una UBF imprecisa respecto a la CR, lo cual no tiene que ver con la codependencia entre salida y coste del Capítulo 12. Esta imprecisión es más importante para CRSs que se originan en algoritmos de divide-y-vencerás, en los que el número de veces que se aplica una ecuación se relaciona con el coste de cada aplicación.

La imprecisión de PUBS se trata en el Capítulo 12 donde se proponen nuevas técnicas precisas para resolver CRs. Éstas se basan en definir la UBF como la solución de una fórmula FOL universalmente cuantificada. Este método puede obtener las UBFs más precisas, pero tiene dos serias limitaciones: (1) hay que proporcionar una UBF plantilla para cada CR; y (2) el uso de la QE para aritmética real no lineal, que es poco escalable, vuelve el método poco útil.

En este capítulo se explora el punto medio entre estos enfoques, y se busca técnicas de resolución que sean tan eficientes como las de PUBS y tan precisas como las del Capítulo 12. En concreto, se describe una nueva técnica en la que

1) se descompone la CR de entrada en varias CRs sencillas, llamadas **atómicas**, 2) se se resuelve cada CR atómica por separado, y 3) se combinan las UBFs de éstas en la UBF de la CR original. En una CR **atómica**, la expresión de coste es 0 en todas las ecuaciones excepto en una, en la que es una expresión de coste básica. Hemos observado que solo hace falta resolver algunas de estas CRs atómicas de manera precisa, usando técnicas como las del Capítulo 12, para obtener una UBF final precisa.

En las Secciones 13.1 y 13.2 se proponen dos métodos para resolver CRs atómicas, que se basan en especificar la UBF con una fórmula FOL universalmente cuantificada, como en el Capítulo 12. Sin embargo, no hace falta que el usuario proporcione plantillas (solo se usan funciones lineales) y, además, las VCs generadas son fórmulas en aritmética lineal, para la cual existen QE métodos muy eficientes. En la Sección 13.3, se muestra cómo manejar el caso general reduciendo el problema de resolver una CR aislada al de resolver varias CRs atómicas. En la Sección 13.5 se describe el trabajo relacionado, y en la Sección 13.6 los contenidos adicionales del artículo. Hemos implementado, dentro de PUBS, nuestras técnicas. Los experimentos muestran que nuestras técnicas son más precisas e tan escalables como las de PUBS.

1 El Método de Suma por Árbol

En esta sección se describe el método de suma-árbol para resolver una CR atómica. Primero se considera el caso en el que la expresión de coste distinta a cero es $\mathsf{nat}(l)$, y al final se ve cómo manejar otras expresiones básicas. Empecemos por un sencillo ejemplo motivador, para el que las otras técnicas de PUBS [5, 14] infieren una UBF imprecisa.

Ejemplo 13.1. Considere la siguiente CR C:

$$\begin{array}{ll} C(m) = 0 & \{m = 0\} \\ C(m) = \mathsf{nat}(n) + C(m-n) & \{m \geq n, n \geq 1\} \end{array}$$

Fíjese que en cada aplicación de la ecuación recursiva añade n (entre 1 y m) unidades al coste, y esta cantidad se resta de la entrada de la llamada recursiva. La Figura 13.1 muestra tres posibles árboles de evaluación para $C(m)$:

- El árbol (a) es el de la cadena de llamadas recursivas más larga posible. En cada nodo, el coste añadido es $n = 1$, lo que permite aplicar la ecuación recursiva m veces. Luego el coste total es m;
- En el árbol (b), el coste local del primer nodo es $n = m$, lo que impide aplicar otra vez la ecuación recursiva. El coste total también es m.

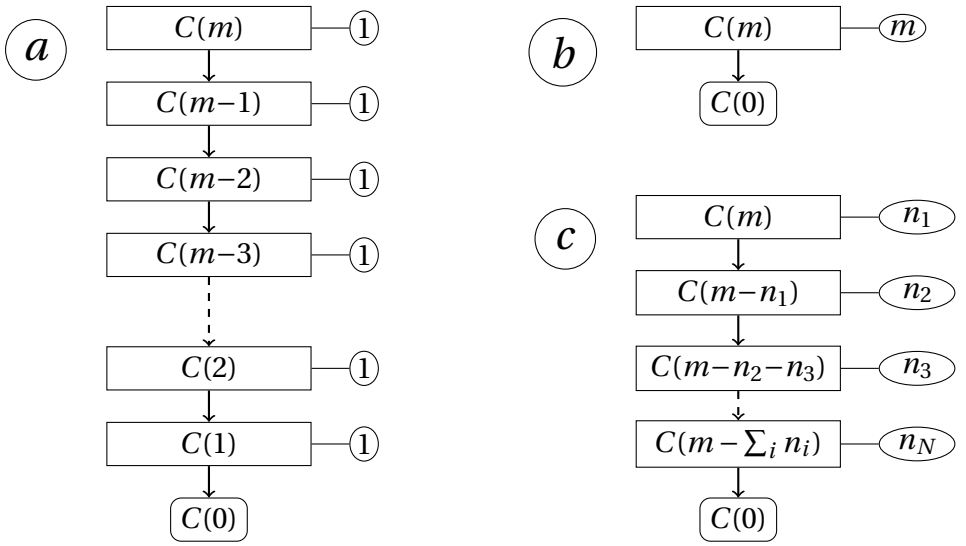


Figura 13.1: Árboles de evaluación para $C(m)$, siendo (a) de longitud máxima, (b) de coste local máximo, y (c) cualquier caso intermedio.

- En el árbol (c), la ecuación recursiva se aplica N veces, para $1 \leq N \leq m$. El nodo-i, que corresponde a la aplicación i-ésima, añade n_i unidades al coste, de tal manera que $m = \sum_{i=1}^{N} n_i$. El coste también es m.

Del árbol (c) se deduce que todas las evaluaciones de $C(m)$ tienen un coste total m. Así, la UBF más precisa es $C^+(m) = \mathsf{nat}(m)$. Sin embargo, PUBS calcula la UBF $C^+(m) = \mathsf{nat}(m)^2$, que es asintóticamente imprecisa, porque multiplica el coste máximo de cada nodo, que es m como en el árbol (b), por el número máximo de nodos internos, que también es m como en el árbol (a). ■

Para manejar con precisión ejemplos como éste, hay que tener en cuenta la relación entre el coste añadido por cada nodo del árbol y el número de nodos en ese árbol. Así se descartarían algunos casos en que el coste local máximo y el número máximo de nodos vienen de árboles distintos. Ninguna de las técnicas en que se basa PUBS [5, 14] puede tomar esta relación en cuenta.

A continuación se describe un método en el que sí se tiene en cuenta esta relación. Éste se basa en el método de aserciones inductivas para demostrar la corrección de programas [34], y se parece al de la Sección 12.2. En concreto, se basa en derivar una VC, en este caso de la CR, cuya validez implique que una UBF candidata es válida.

Ejemplo 13.2. Dada la CR del ejemplo 13.1 y una UBF candidata $C^+(m)$, la VC es una conjunción de estas cláusulas:

$$\begin{array}{lll} \forall m: & m = 0 & \rightarrow \quad C^+(m) \geq 0 \\ \forall m, n: & m \geq n \geq 1 & \rightarrow \quad C^+(m) \geq n + C^+(m-n) \end{array}$$

Cada cláusula corresponde a una ecuación de la CR, y afirma que la UBF C^+ cubre el coste local de la ecuación correspondiente más el coste de las llamadas recursivas. Esta VC es similar a las del coste neto de la Sección 12.2. ■

Para comprobar si una expresión de coste $f(m)$ es una UBF válida, solo hay que ponerla en el lugar de $C^+(m)$ en la VC, y comprobar si ésta es válida. Sin embargo, como en la Sección 12.2, nos interesa más sintetizar una UBF que comprobar si una es válida. Como en Capítulo 12, para esto se usan UBFs plantilla y resolutores de QE y SMT.

Ejemplo 13.3. Sea $f(m) = \mathsf{nat}(a_\mathrm{m} m + a_\mathrm{c})$ una UBF plantilla para la CR del ejemplo 13.1, siendo a_m y a_c sus parámetros. Al ponerla en la VC del ejemplo 13.2 se obtiene una fórmula FOL que es la conjunción de estas cláusulas:

$$\begin{array}{lll} \forall m: & m = 0 & \rightarrow \mathsf{nat}(a_\mathrm{m} m + a_\mathrm{c}) \geq 0 \\ \forall m, n: & m \geq n \geq 1 & \rightarrow \mathsf{nat}(a_\mathrm{m} m + a_\mathrm{c}) \geq \mathsf{nat}(n) + \mathsf{nat}(a_\mathrm{m}(m-n) + a_\mathrm{c}) \end{array}$$

en las que a_m y a_c son variables libres. Cualquier solución de esta fórmula da una instancia de la plantilla que es una UBF válida. Por ejemplo, $a_\mathrm{m} = 1, a_\mathrm{c} = 0$ define la UBF $f(m) = \mathsf{nat}(m)$. Para encontrar tales instancias automáticamente, primero se aplica un procedimiento de QE para quitar las variables universalmente cuantificadas n y m. Ello da la restricción equivalente $a_\mathrm{m} \geq 1 \wedge a_\mathrm{c} \geq 0$. ■

Como se dijo en el Capítulo 12, para algunas teorías FOL los procedimientos QE existentes son computacionalmente caros. No obstante, si nos restringimos a UBFs plantillas de la forma $\mathsf{nat}(l)$, en la que l una expresión lineal con coeficientes paramétricos, como la del Ejemplo 13.3, entonces se obtiene una VC cuya forma es casi lineal, dado que nos hemos restringido a CRs atómicas con una sola expresión $\mathsf{nat}(l')$. Por suerte, para resolver una VC de esta forma sí hay una técnica eficiente basada en programación lineal (de complejidad polinomial), a diferencia de las que se usaba en el prototipo del Capítulo 12. Esto hace que nuestro método sea útil. El apéndice del artículo (Página 268) detalla cómo resolver estas VCs.

Manejar otras Expresiones Básicas. Ahora se extiende esta técnica para manejar cualquier expresión básica b, sea $b = m^{\mathsf{nat}(l)} - 1$ o $b = \log_m(\mathsf{nat}(l) + 1)$, en vez de solo $b = \mathsf{nat}(l)$. Fíjese que todavía hace falta que solo una ecuación tenga un coste local distinto a cero. Una posible solución sería generar la VC con estas expresiones básicas, pero la VC así obtenida no tendría la forma deseada, y resolverlas podría ser computacionalmente caro. En vez de eso, para resolver estas CRs, se reemplaza la expresión de coste básica ($m^{\mathsf{nat}(l)} - 1$ o $\log_m(\mathsf{nat}(l) + 1)$) por la subexpresión $\mathsf{nat}(l)$, lo que da una CR E. Ésta se puede resolver, tal y como antes se ha descrito, en una UBF $E^+(\bar{x}) = \mathsf{nat}(L)$. Con ésta se puede generar

una UBF para C de este modo:

$$C^+(\bar{x}) = \begin{cases} 1{,}5 * \mathsf{nat}(L) & \text{if } b = \log_m(\mathsf{nat}(l)+1) \\ m^{\mathsf{nat}(L)} - 1 & \text{if } b = m^{\mathsf{nat}(l)} - 1 \end{cases}$$

Ejemplo 13.4. Considere la CR C definida con estas ecuaciones:

$$\begin{array}{ll} C(m) = 0 & \{m = 0\} \\ C(m) = 2^{\mathsf{nat}(n)} - 1 + C(m-n) & \{m \geq n, n \geq 1\} \end{array}$$

Se reemplaza $2^{\mathsf{nat}(n)} - 1$ por $\mathsf{nat}(n)$, lo que da la siguiente CR:

$$\begin{array}{ll} E(m) = 0 & \{m = 0\} \\ E(m) = \mathsf{nat}(n) + E(m-n) & \{m \geq n, n \geq 1\} \end{array}$$

Esta CR se resuelve como antes a la UBF $E^+(m) = \mathsf{nat}(m)$. Entonces se obtiene $C^+(m) = 2^{\mathsf{nat}(m)} - 1$ como la UBF de C. Fíjese que, de haber usado PUBS, se habría obtenido la UBF $C^+(m) = \mathsf{nat}(m) * (2^{\mathsf{nat}(m)} - 1)$ que es imprecisa. ■

2 El Método de Suma por Niveles

La técnica de la sección anterior maneja casos en que la CR atómica acepta una UBF lineal. En esta sección se describe un método similar para resolver CRs atómicas del tipo *divide-y-vencerás*, que no admiten una UBF lineal. De nuevo, se supone una expresión básica de la forma $\mathsf{nat}(l)$, y luego se ve cómo manejar otras expresiones básicas. Empecemos por mostrar el ejemplo de motivación para esta sección.

Ejemplo 13.5. Considere una CR C definida con estas ecuaciones:

$$\begin{array}{lll} C(m) = & 0 & \{m \geq 0\} \\ C(m) = & \mathsf{nat}(m) + C(m_1) + C(m_2) & \{m = m_1 + m_2 + 1, m_1 \geq 0, m_2 \geq 0\} \end{array}$$

Fíjese que cada aplicación de la ecuación recursiva añade un coste igual a la entrada m, y que sus llamadas recursivas se aplican a m_1 y m_2 donde $m = m_1 + m_2 + 1$. Éste es un caso típico de divide-y-vencerás.

Según el valor de m_1 y m_2 en cada aplicación recursiva, la llamada a $C(m)$ puede evaluarse a distintos valores. Por ejemplo, si en cada aplicación se escoge $m_1 = m-1$ y $m_2 = 0$, se obtiene el coste en el caso peor $\frac{m^2+m}{2}$, como muestra el árbol (a) de la Figura 13.2. Por otro lado, si se divide $m-1$ por igual entre m_1 y m_2 se obtiene el caso mejor que está en $\Theta(m \log_2 m)$, como muestra el árbol (b) de la Figura 13.2. Para este ejemplo PUBS infiere la UBF $C^+(m) = \mathsf{nat}(m) * (2^{\mathsf{nat}(m)} - 1)$ que es imprecisa. ■

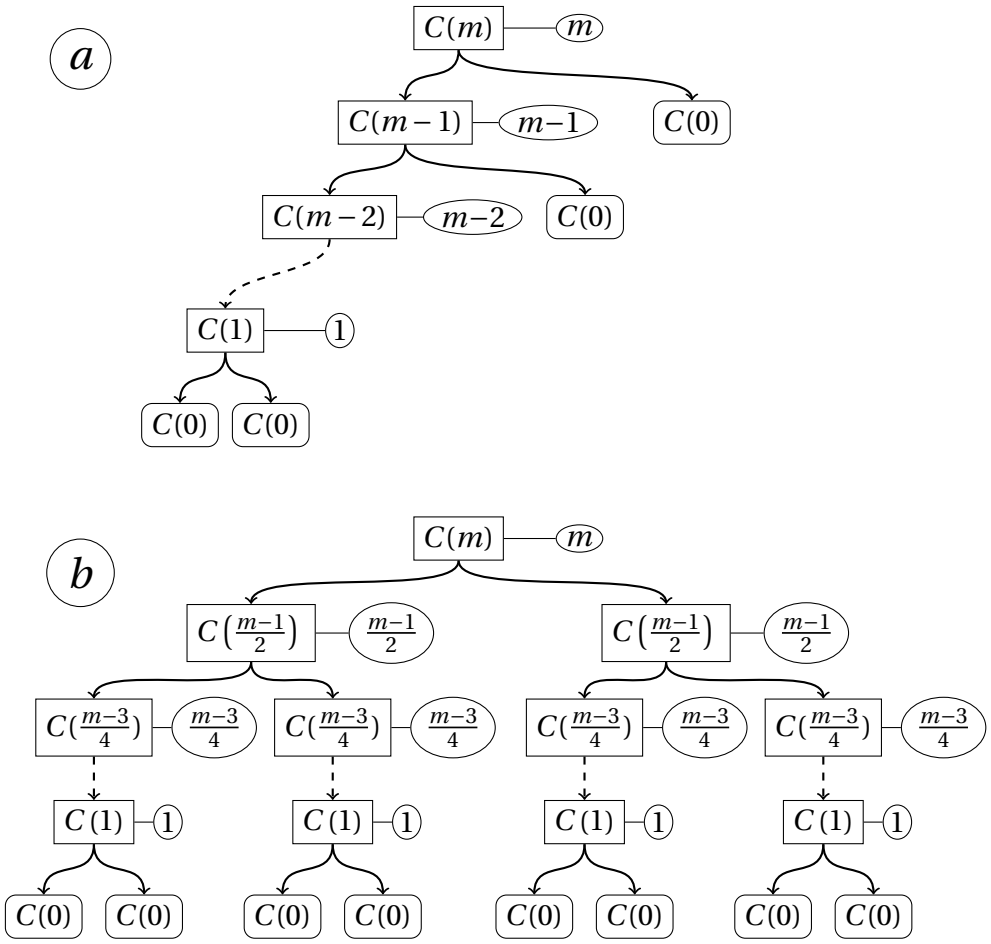


Figura 13.2: Árboles de evaluación para la CR $C(m)$.

Como muestra el árbol de evaluación (a) de la Figura 13.2, el coste de esta CR no es lineal. Por ello, para resolver esta CR con el método de suma por árbol habría que usar una UBF plantilla polinomial, y si no fallaría. Sin embargo, solo se quiere usar UBF plantilla lineales, para que la VC se pueda resolver eficazmente. Por ello, no se puede usar el método de suma por árboles. En cambio, a continuación se describe un método alternativo para resolver estas CRs.

La intuición de este método es que en un ejemplo de divide-y-vencerás, la expresión de coste $w(\bar{x}) * h(\bar{x})$ es una UBF de la CR si (i) $w(\bar{x})$ acota el coste total de cada nivel de todo árbol de evaluación; y (ii) $h(\bar{x})$ acota la altura de todo árbol. Por ejemplo, para la CR del ejemplo 13.5 sirven $w(m) = \mathsf{nat}(m)$ y $h(m) = \mathsf{nat}(m)$, lo que da la UBF $C^+(m) = \mathsf{nat}(m)^2$. Así, el método infiere $w(\bar{x})$ y $h(\bar{x})$ por separado, y los multiplica en la UBF de la CR. La ventaja es que ambas cotas son lineales, y por tanto se pueden sintetizar de manera eficiente. El próximo

ejemplo describe de manera básica este método, al derivar las VCs y sintetizar $w(m)$ and $h(m)$ para la CR del ejemplo 13.5.

Ejemplo 13.6. Considere la CR del ejemplo 13.5 y, por concisión, sea φ_2 las restricciones de la ecuación recursiva. Una expresión de coste $h(m)$ es una cota en la altura si la siguiente VC es válida.

$$\forall m, m_1, m_2 : \varphi_2 \quad \rightarrow \quad h(m) \geq 1 + h(m_1) \wedge h(m) \geq 1 + h(m_2)$$

Intuitivamente, puesto que la altura es el número de aplicaciones consecutivas de la ecuación recursiva, la VC exige que $h(m)$ cubra una aplicación de esta ecuación y todas las que pueda haber en cada llamada recursiva. Fíjese que cada llamada se considera por separado (la cláusula tiene una conjunción), puesto que corresponden a ramas distintas del árbol. Además, es necesario que $h(m)$ sea no negativa, o esta VC no sería correcta.

Asimismo, una expresión de coste $w(m)$ es una costa en el coste de cada nivel si la VC definida como la conjunción de estas cláusulas es válida:

$$\begin{array}{lll} \forall m, m_1, m_2 : \varphi_2 & \rightarrow & w(m) \geq \mathsf{nat}(m) \\ \forall m, m_1, m_2 : \varphi_2 & \rightarrow & w(m) \geq w(m_1) + w(m_2) \end{array}$$

Intuitivamente, la primera cláusula exige que $w(m)$ cubra el coste contribuido por la ecuación recursiva en cada nivel, y la segunda clausula que también cubra el siguiente nivel. También aquí importa que $w(m)$ sea una expresión de coste, lo que asegura que no es negativa.

Para calcular $h(m)$ y $w(m)$, se usan las funciones plantilla $h(m) = \mathsf{nat}(h_{\mathrm{m}} m + h_{\mathrm{c}})$ y $w(m) = \mathsf{nat}(w_{\mathrm{m}} m + w_{\mathrm{c}})$, y se aplica QE para eliminar las variables universalmente cuantificadas (las de la CR). Así se obtienen las restricciones $h_{\mathrm{m}} \geq 1 \wedge h_{\mathrm{c}} + h_{\mathrm{m}} \geq 1$ y $w_{\mathrm{m}} \geq 1 \wedge w_{\mathrm{m}} \geq w_{\mathrm{c}} \wedge w_{\mathrm{m}} + w_{\mathrm{c}} \geq 1$, que admiten las soluciones respectivas $h_{\mathrm{m}} = 1, h_{\mathrm{c}} = 0$ y $w_{\mathrm{m}} = 1, w_{\mathrm{c}} = 0$. Así pues, $h(m) = \mathsf{nat}(m)$ y $w(m) = \mathsf{nat}(m)$ son cotas válidas en la altura y en el coste de cada nivel. ■

En una CR atómica puede haber ecuaciones recursivas sin coste, esto es, con coste local cero. En este caso, algunos árboles del árbol de evaluación pueden tener coste 0, y sería más preciso no contar éstos en $h(\bar{x})$. Para ello, en vez de acotar la altura del árbol, se acota el número de nodos con coste distinto a cero en cada rama del árbol. Asimismo, en vez de tomar el nivel como la distancia a la raíz, se toma como el número de nodos con coste no nulo desde la raíz. Esto se formaliza en el artículo.

Manejar Expresiones de coste básicas. Ahora se extiende la técnica a otras expresiones de coste básicas b, sean $b = m^{\mathsf{nat}(l)} - 1$, o $b = \log_m(\mathsf{nat}(l) + 1)$, en vez de solo $b = \mathsf{nat}(l)$. Fíjese que aun hace falta que solo una ecuación tenga

un coste no nulo. En el método de suma por niveles hay dos VCs, la que corresponde a $h(\bar{x})$ es la misma pues la altura no depende de b. La VC para $w(\bar{x})$ se hace tal y como se describe al final de la sección anterior.

3 Resolver el Caso General de CRSs

En esta sección se muestra cómo resolver cualquier CR aislada. El caso general para resolver un CRS se hace como se describe en la Sección 9.5.

Los métodos de suma-árbol y suma por niveles se formulan para resolver una CR atómica, en la que el coste es nulo en todas las ecuaciones excepto en una, en la que es una expresión básica; en cambio, en una CR aislada puede haber varias ecuaciones en las que el coste no sea nulo y además sea cualquier expresión de coste. A continuación vemos cómo manejar tales CRs. La idea básica es romper la CR de entrada en varias CRs atómicas, resolver éstas a UBFs usando los métodos de suma por árbol o por niveles, y combinar estas UBFs en una UBF de la CR original. Nuestro enfoque se presenta en dos pasos: 1) se extienden las CRs atómicas para permitir productos de expresiones básicas, y 2) se trata el caso general en que varias ecuaciones tienen un coste no nulo. Por simplicidad, suponemos que la CR no usa el operador máx. Si no, éste se reemplaza por una suma de los argumentos, como en la Sección 10.2, o bien se duplica la ecuación de coste una vez por cada argumento del operados máx. También suponemos que las expresiones de coste están en forma normal, esto es, como sumas de productos de expresiones básicas.

Productos. Supongamos una CR C en la que el coste es nulo en todas las ecuaciones excepto en una, en la que el coste es un producto $e = b_1 * \cdots * b_n$ de expresiones básicas b_i. La CR C se resuelve del siguiente modo:

1. Se escoge una b_i, con $1 \leq i \leq n$, y se reemplaza e por b_i, obteniéndose una CR E atómica; ésta se resuelve en una UBF $E^+(\bar{x})$ usando los métodos de las secciones previas;
2. para cada b_j con $j \neq i$ se calcula su maximización $\hat{b}_j$ (véase Sección 9.5), y se construye $C^+(\bar{x}) = E^+(\bar{x}) * \prod_{j \neq i} \hat{b}_j$ como UBF de C.

El método es correcto ya que, siendo $\hat{b}_j$ mayor que cualquier instancia de b_j y constante en el árbol (solo depende de los parámetros de la llamada inicial), se puede factorizar fuera de la CR. Nótese que si en el primer paso se escoge una i distinta, se puede obtener otra UBF.

Ejemplo 13.7. Considere la CR C definida con estas ecuaciones:

$$\begin{array}{ll} C(m,p) = 0 & ,\{m = 0\} \\ C(m,p) = \mathsf{nat}(q) * \mathsf{nat}(n) + C(m-n,p') & ,\{m \geq n \geq 1, p \geq q, p \geq p' \geq 0\} \end{array}$$

El coste de esta segunda ecuación es un producto de expresiones básicas. Si éste se reemplaza por $\mathsf{nat}(n)$, se obtiene esta CR atómica:

$$\begin{array}{ll} E(m,p) = 0 & ,\{m = 0\} \\ E(m,p) = \mathsf{nat}(n) + E(m-n,p') & ,\{m \geq n \geq 1, p \geq q, p \geq p' \geq 0\} \end{array}$$

E se puede resolver en la UBF $E^+(m,p) = \mathsf{nat}(m)$. El otro factor $\mathsf{nat}(q)$ se maximiza a $\mathsf{nat}(p)$, siendo p el parámetro de la CR original, y por último se define $C^+(m,p) = \mathsf{nat}(p) * \mathsf{nat}(m)$ como UBF de la CR C. ■

El caso general. Supongamos una CR C con k ecuaciones, donde la ecuación i tiene una expresión de coste $e_i = P_{i1} + \cdots + P_{in_i}$, y cada sumando P_{ij} es un producto de expresiones básicas. C se resuelve en una UBF del siguiente modo:

1. Para cada producto P_{ij}, se define la CR E_{ij} que se obtiene al quitar todos los otros productos de C, y cada E_{ij} se resuelve en una UBF $E^+_{ij}(\bar{x})$.
2. Se construye la UBF $C^+(\bar{x})$ de $C(\bar{x})$ as $C^+(\bar{x}) = \sum_{i=1}^{k} \sum_{j=1}^{n_i} E^+_{ij}(\bar{x})$.

Este método es correcto en tanto que cada CR E_{ij} modela la contribución del sumando P_{ij} al coste total de C.

Ejemplo 13.8. Considere la CR C definida con éstas ecuaciones:

$$\begin{array}{lll} \text{Eq,1} & C(m,n) = 0 & ,\varphi_1 \\ \text{Eq,2} & C(m,n) = \underbrace{\mathsf{nat}(n)}_{P_{21}} + C(m,n_1) & ,\varphi_2 \\ \text{Eq,3} & C(m,n) = \underbrace{\mathsf{nat}(m_0)}_{P_{31}} + \underbrace{\mathsf{nat}(n)}_{P_{32}} + C(m_1,n_1) + C(m_2,n_2) & ,\varphi_3 \end{array}$$

siendo $\varphi_1 \equiv \{m_0 = 0, n = 0\}$, $\varphi_2 \equiv \{n \geq 1, n = 1+n_1\}$, y $\varphi_3 \equiv \{m \geq m_0+m_1+m_2, n \geq 1+n_1+n_2\}$. En esta CR hay tres productos (de un solo factor): $P_{21} = \mathsf{nat}(n)$, $P_{31} = \mathsf{nat}(m_0)$, and $P_{32} = \mathsf{nat}(n)$. Para resolver C, ésta se descompone en las CRs E_{21}, E_{31}, y E_{32} en la Figura 13.3; éstas se resuelven en las UBFs $E^+_{21}(m,n) = \mathsf{nat}(n)^2$, $E^+_{31}(m,n) = \mathsf{nat}(m)$, y $E^+_{32}(m,n) = \mathsf{nat}(n)^2$. Entonces se construye la suma $C^+(m,n) = \mathsf{nat}(n)^2 + \mathsf{nat}(m) + \mathsf{nat}(n)^2$ como la UBF de $C(m,n)$. ■

4 Evaluación Experimental

Hemos implementado nuestras técnicas como una extensión de PUBS. La implementación ocupa menos de 750 líneas de código PROLOG.

$$
\begin{array}{lll}
E_{21}(m,n) = & 0 & ,\varphi_1 \\
E_{21}(m,n) = & \mathsf{nat}(n)+ \quad E_{21}(m,n_1) & ,\varphi_2 \\
E_{21}(m,n) = & 0+ \quad E_{21}(m_1,n_1)+E_{21}(m_2,n_2) & ,\varphi_3 \\
E_{31}(m,n) = & 0 & ,\varphi_1 \\
E_{31}(m,n) = & 0+ \quad E_{31}(m,n_1) & ,\varphi_2 \\
E_{31}(m,n) = & \mathsf{nat}(m_0)+ \quad E_{31}(m_1,n_1)+E_{31}(m_2,n_2) & ,\varphi_3 \\
E_{32}(m,n) = & 0 & ,\varphi_1 \\
E_{32}(m,n) = & 0+ \quad E_{32}(m,n_1) & ,\varphi_2 \\
E_{32}(m,n) = & \mathsf{nat}(n)+ \quad E_{32}(m_1,n_1)+E_{32}(m_2,n_2) & ,\varphi_3
\end{array}
$$

Figura 13.3: Descomposición de una CR en tres CRs menores.

Precisión. Para evaluar la precisión asintótica de nuestras técnicas, hemos escrito varios programas con ejemplos de la literatura. Aunque estos programas eran cortos, su análisis de coste planteaba algunos retos que se resuelven por nuestros métodos. En concreto, para el método de suma por árbol se usan algunos programas que se suelen describir en términos de coste amortizado, como el procedimiento para añadir varios elementos en una lista implementada con un array expandible [45, §17], las operaciones para añadir y quitar elementos de una cola construida con dos pilas [103]. Para el método de suma por niveles se usan algunos algoritmos de *divide y vencerás*, como QuickSort. En el artículo se muestra para cada ejemplo la AUBF que COSTA obtiene, usando el método para resolver CRs de [5]; y la AUBF and se obtiene usando nuestros métodos. Para todos los ejemplos, los métodos de [5] obtienen una AUBF imprecisa, mientras que nuestros métodos obtienen una AUBF precisa para el programa. Aunque PUBS ya tiene una técnica para resolver CRs del estilo divide-y-vencerás [5], ésta era de aplicabilidad restringida, y por ejemplo no podía usarse para resolver la CR de QuickSort.

Escalabilidad Siguiendo la técnica de [5, §10], hemos mezclado los programas de los experimentos anteriores para construir un conjunto de programas de tamaño creciente como suite de *benchmarks*. Para cada uno, se mide el tiempo que se tarda en resolver el CRS con los métodos de PUBS y con los nuestros. En los experimentos, vimos que los tiempos de nuestro método son razonables, y en una proporción constante respecto a los de los tardan los métodos de [5].

También hemos comparado nuestro enfoque con el prototipo ACRP de [18]. ACRP no pudo obtener una UBF en menos de un minuto para ningún benchmarks; lo cual era de esperar ya que ACRP usa un procedimiento de QE para aritmética real no lineal que no es escalable. En cambio, nuestros métodos resuelven todos los programas en unos segundos. Ello se debe al uso de un mé-

todo de QE para aritmética lineal real, basado en programación lineal.

En síntesis, en nuestros experimentos se ve que nuestro método es más preciso que [10], y que su escalabilidad es similar a la de los métodos existentes de PUBS.

5 Trabajo Relacionado

Resolución de Relaciones de Recurrencia. Los primeros métodos automáticos para resolver relaciones de recurrencia a forma cerrada se desarrollaron para los sistemas de álgebra por computador como MACSYMA [78] y MATHLAB [42]. PURRS [27] es un sistema reciente, que también intenta resolver relaciones de recurrencia, e intenta aproximar la solución cuando no puede encontrar una exacta. Estos sistemas asumen recurrencias deterministas, que no pueden modelar el coste de programas abstractos indeterministas.

PUBS. Los trabajos más cercanos al nuestro son los que describen PUBS. La sintaxis y semántica de los CRSs, y su diferencia con las relaciones de recurrencia, se describen en [5]. Este artículo también describe los métodos para resolver CRSs: uno es del *node-count*, descrito en la Sección 9.5; el otro es el del *level-count*, para resolver CRSs que surgen de algoritmos de divide-y-vencerás, como nuestro método de suma por niveles aunque de una aplicabilidad más restringida. PUBS también usa las técnicas de [14], que resuelven CRSs transformándolos primero a una relación de recurrencia que aproxima el caso peor, y luego resuelve ésta con un sistema de álgebra por computador. Este método obtiene UBFs más precisas que las de [5], y también se puede usar para inferir LBFs. Sin embargo, su aplicabilidad es más restringida, y para algunos ejemplos también infieren una UBF asintóticamente imprecisa.

Las expresiones de coste se introducen en [5], como una forma de expresiones monótonas construidas a partir de expresiones lineales. La clave de las expresiones de coste es que, al ser monótonas en las componentes nat, una expresión de coste se maximiza maximizando las expresiones lineales en las expresiones nat. Así, las expresiones de coste son una palanca, para razonar sobre expresiones complejas usando técnicas de programación lineal. Del mismo modo, nuestra descomposición de una CR en CRs atómicas es un palanca para resolver una CRs compleja usando los métodos de QE para aritmética lineal.

Razonamiento lógico y Eliminación de Cuantificadores. Las VCs para los métodos de suma por árbol y por niveles se basan en el método de aserciones inductivas para probar corrección de programas [34]. El uso de QE para aritmética lineal se ha usado en varios trabajos relacionados, para inferir funciones

de rango lineales [26], y lexicográficas [15]. En [124] se usan aserciones inductivas con plantillas para la verificación y síntesis de programas. Su herramienta maneja una clase amplia de predicados plantilla, que incluyen disyunciones y conjunciones de restricciones lineales.

Cotas de Alcanzabilidad. La cota en la altura de los árboles de evalación, que se usa en el método de suma por niveles, se puede ver como el número de visitas secuenciales a una ecuación, lo que se conoce como una cota de alcanzabilidad (*reachability bound* [66]). Ésta se puede calcular sobre una función de rango lineal [26] o lexicográfica [15]. En [66], se define este problema y, para calcular esas cotas, se presentan técnicas basadas en SMT que permiten calcular UBFs que son productos de máximos de expresiones `nat`. En [140] se combina ese enfoque con la abstracción de cambio de tamaño (*size-change*). En otros trabajos se propone inferir funciones de rango polinómicas. En [38], proponen usar métodos de QE métodos para aritmética real no lineal. Cousot [47] usa relajación lagrangiana para reducir el problema de la QE como expresiones polinómicas a uno de programación lineal. Otras técnicas se proponen en [97].

6 Contenido Adicional

En este capítulo se describe las contribuciones de [17]. Se ha explicado cómo se resuelve una CR aislada descomponiéndola a varias CRs atómicas, cómo éstas se resuelven a sus UBFs, y cómo estas UBFs se combinan en una UBF para la CR original. Para resolver CRs atómicas, se han explicado los métodos de suma por árbol y de suma por niveles, basados en el uso de fórmulas FOL cuantificadas como en la Sección 12.2, pero que se resuelven eficientemente usando QE para aritmética lineal.

El artículo extiende y formaliza el contenido de este capítulo. Contiene una definición formal de la clase sintáctica de CRs atómicas. También formaliza el método de suma por árbol y el método de suma por niveles, y proporciona lemas de su corrección junto con sus demostraciones. El artículo describe el procedimiento de QE que se usa para resolver las VCs, el cual se basa en el uso del Lema de Farkas, y cómo se adapta para manejas las expresiones `nat` en las VCs. El artículo también contiene las tablas con los resultados de nuestros experimentos.

14 | Conclusiones y Trabajo Futuro

En este capítulo se exponen las conclusiones de esta tesis. En la Sección 14.1 se describen los logros, el impacto, así como la aplicabilidad, de nuestras contribuciones. En la Sección 14.2 se comenta, aparte, nuestras conclusiones sobre la relación entre los enfoques clásico y amortizado al análisis de coste. Por último, en la Sección 14.3 dibujamos algunas líneas para posible trabajo futuro.

1 Objetivos y Logros

El primer objetivo de esta tesis era adaptar los resolutores existentes de CRSs para que calculen UBFs asintóticas, al ser éstas más adecuadas para especificar el coste durante el desarrollo de un programa. En particular, las asintóticas son: (I) menos sensibles a pequeños cambios en el programa, y (II) más concisas y legibles. Esto se logra en el Capítulo 10, en dos pasos siguientes:

1. Siguiendo un enfoque transformacional, se ha definido un método para transformar UBFs a forma asintótica. Ello tiene la ventaja de poder aplicarse directamente a las UBFs que obtiene cualquier analizador, no solo COSTA, pero la desventaja de que para calcular una UBF asintótica antes hay que calcular una no asintótica.
2. Se construye un resolutor de CRSs que directamente calcula una UBF asintóticas. Para ello, se intercala el método anterior con las fases de PUBS. Tal resolutor puede usarse en cualquier análisis de coste que genere CRSs, y es más escalable que el primer método.

A diferencia de otros métodos anteriores [126], en nuestro método se usa la información de contexto para obtener una cota asintótica más concisa.

El segundo objetivo era explorar la diferencia de precisión entre los distintos enfoques de análisis de coste. Para ello había que (I) entender por qué razón en COSTA, y en general en el enfoque clásico de Wegbreit, se infieren UBFs imprecisas para algunos programas; e (II) inventar técnicas para resolver esta imprecisión. En este sentido, se han identificado varias causas de esa imprecisión, asociadas a varias fases de COSTA, y se ha propuesto cómo remediarlas:

- En el Capítulo 11 se resuelve la imprecisión que causa el uso de operaciones aritméticas no lineales, que no se pueden modelar en COSTA pues en éste se usan restricciones lineales. Se ve cómo, usando la información del contexto de la operación, una operación no lineal se puede modelar usando restricciones lineales, las cuales pueden codificarse directamente en el programa ACR para así no tener que cambiar el analizador. Con este mé-

todo, se logra que COSTA maneje programas que antes no podía.
- En el Capítulo 12 se ve que hay programas en los que emerge una relación entre la salida y el coste de un procedimiento, y cómo esta relación causa que al analizar estos programas aparezcan problemas de imprecisión. En concreto, se ve que éstos surgen en la fase que transforma el programa ACR en un CRS, al quitar los parámetros de salida. Para resolverlos, se desarrollan nuevas técnicas de análisis, basadas en especificar el coste como una fórmula FOL sobre algunos UBF plantilla, y usar métodos de QE y SMT. Estas técnicas se extienden para modelos de coste que representan recursos que se liberan, en los que aparece la noción de coste pico. Estos resultados tienen una importancia teórica para el análisis de coste, si bien la mayoría de procedimientos de QE son computacionalmente caros, lo que hace que este método no escale para grandes programas.
- Para lograr la escalabilidad deseada, esas técnicas se combinan en el Capítulo 13 con las de PUBS. Para ello, la CR de entrada se descompone en varias CRs atómicas; cada una de éstas se resuelve en una UBF, para lo cual se deriva una fórmula FOL que se resuelve usando un procedimiento de QE eficiente, basado en programación lineal. Con esta técnica se auna la mejora en precisión que se logra en el Capítulo 12 con la escalabilidad que se logra en PUBS.

En síntesis: además de justificar, por primera vez, la causa de los problemas de imprecisión que aparecen en el enfoque clásico al análisis de coste, nuestra mayor contribución es aplicar *El Cálculo de la Informática* (*The Calculus of Computation* [34]) al análisis estático de coste, al ver la inferencia de UBFs y la resolución de CRSs como la verificación de programas abstractos, y así aprovechar los avances recientes en resolución de SMT [89] y QE [81].

Nuestras contribuciones tienen un impacto fundamental en el análisis de coste, tanto en los aspectos teóricos como en el aspecto prácticos. En éste, el impacto se ve en la funcionalidad que se ha añadido a COSTA, así como en las mejoras en su aplicabilidad, precisión, y escalabilidad.

- Nuestra primera contribución extiende la funcionalidad de COSTA, haciendo posible inferir cotas asintóticas. Como se ve en la Sección 10.5, la resolución asintótica de CRSs mejora la escalabilidad de COSTA, y de este modeo mejora su aplicabilidad a programas mayores.
- Nuestra segunda contribución extiende la aplicabilidad de COSTA a programas JBC cuyo coste depende de una instrucción no lineal. Como se ve en la Sección 11.2, esta contribución permite a COSTA inferir relaciones de valor más fuertes, con las que PUBS puede inferir AUBFs más precisas. Además, esto se logra sin dañar la escalabilidad de COSTA, no solo porque se usa un dominio abstracto no disyuntivo, sino también porque no se au-

menta el tamaño del CRS generado.

- Nuestra tercera contribución (Capítulo 12) tiene un impacto que trasciende de sus aspectos prácticos: proyecta una nueva luz sobre la relación entre los varios enfoques de análisis de coste. Por el lado práctico, hemos implementado nuestro enfoque en un prototipo llamado ACRP, descrito en la Sección 12.5. ACRP consigue inferir una AUBF precisa para muchos programs para los que COSTA falla, entre ellos algunos ejemplos típicos de análisis amortizado, así como algunos bucles anidados. Sin embargo, ACRP usa los métodos de SMT y QE para la teoría FOL de campos reales (números reales), los cuales no son escalables. Por ello, ACRP no tiene un impacto directo en COSTA, y por ello no se le ha integrado en éste.
- Nuestra cuarta contribución mejora la precisión de PUBS, al desarrollar un nuevo método para resolver una CR en una UBF. Como se ve en la Sección 13.4, nuestra técnica logra mejor precisión que las que obtienen las técnicas de [5, 14]. Al basarse en un método de QE para aritmética real lineal, nuestro método mantiene una escalabilidad similar a la de COSTA. Además, nuestro método extiende la aplicabilidad de COSTA y de PUBS a algunos programas que no admiten una función de rango lineal, incluso a algunos programas no terminantes, ya que nuestro método no necesita calcular una función de rango lineal.

Para terminar la discusión del impacto, además de las evidentes mejoras para COSTA (véase secciones 10.5, 11.2, 12.5, y13.4), creemos que las contribuciones teóricas del capítulo 12 hacen más fácil transferir entre los distintos enfoques de análisis de coste el conocimiento y las técnicas de implementación.

Aplicabilidad de las Contribuciones

Las contribuciones de esta tesis, como se han desarrollado al nivel de la ACR y del CRS, son genéricas e independientes respecto al lenguaje de programación, modelo de coste, y abstracción de tamaño, que solo se tratan en la primera fase. Por ello, también se pueden aplicar con un modelo de coste no acumulativo, como el consumo de memoria [13] o de tareas concurrentes [11], así como para cualquier modelo de coste que quiera definir el programador [99, 98]. Asimismo, nuestras contribuciones también son aplicables si las variables representan medidas de tamaño para estructuras de datos dinámicas [123], o medidas de tamaño que dependan del tipo [73], o cualquier medida de tamaño que defina el usuario [65]. Respecto al lenguaje de programación, la compilación abstracta (Sección 9.3) se puede adaptar para analizar datos almacenados en los campos de estructuras de datos [6]. Nuestras contribuciones son aplicables al analizador COSTABS [138] para el lenguaje concurrente ABS.

2 Análisis Amortizado de Coste

Cuando comenzamos nuestra investigación, estudiamos varios programas para los que COSTA infería una UBF imprecisa: había programas (funcionales) para los que los métodos de análisis amortizado de coste [79, 73] sí inferían una UBF precisa; había bucles anidados para los que los métodos de [64, 66] inferían una UBF precisa, a la que llamaban la complejidad *amortizada*; y había también estructuras de datos para lenguajes imperativos [45, §17]. Viendo que estos programas suelen describirse en términos de coste amortizado, intentamos usar las nociones del análisis amortizado para averiguar las causas de la imprecisión.

Lamentablemente, no encontramos una definición clara de coste amortizado en la literatura. En el artículo de Tarjan [129] se describen dos métodos manuales para el análisis de coste, el del banquero y el del potencial, y éstos se usan para analizar programas que realizaban secuencias de operaciones en algunas estructuras de datos. En ese artículo se usa la palabra *amortización* para explicar por qué el foco del análisis se puede mover de el coste de una operación al coste de una secuencia de éstas. En este sentido, el término *coste amortizado* solo es una metáfora para referir el resultado de analizar el coste en el caso pero de esos programas usando dichos métodos; pero en algunos trabajos [103] se usa ese término en contraposición al de *coste en el caso peor*, como si se refirieran propiedades semánticas distintas, lo que causa confusión.

Creemos que con nuestras contribuciones se aclara la noción de coste amortizado. En concreto, se identifica con exactitud qué abstracciones de COSTA añaden imprecisión en el análisis de esos programas:

- Por un lado, en la Sección 12.1 se describe la codependencia entre salida y coste, y cómo esta provoca que COSTA infiera una UBF asintóticamente imprecisa para algunos programas, tales como los bucles anidados de [64, 66], o los métodos de un contador binario [45, §17].
- Por otro lado están esos programas, también asociados en la literatura con la noción de coste amortizado, en los que esta codependencia no se da. Dicho en términos de COSTA, para éstos hay un CRS que representa su coste de manera precisa. Entre ellos están los métodos que operan sobre una tabla dinámica [45, §17.4], o sobre una implementación de una cola con dos pilas [103, §3]. Para estos programas, la imprecisión de COSTA se da al resolver un CRS, en concreto, en que PUBS no considera las dependencias entre los costes de las distintas iteraciones de un bucle, como se ve en el Capítulo 13.

Esta tesis es el primer trabajo en presentar estas observaciones, y en distinguir las causas de *amortización de coste*.

3 Trabajo Futuro

Estamos considerando varias líneas de investigación, con el fin de mejorar las contribuciones de esta tesis.

- En el Capítulo 13 se mejora la precisión asintótica de la resolución del CRS. Como trabajo futuro se podría estudiar cómo mejorar la precisión no asintótica de las UBFs calculadas, especialmente para aquellas CRs con varias ecuaciones o con expresiones máx. Para ello se podría usar el método de optimización de SPEED [65].
- Esta contribución también se podría extender para resolver CRs más diversas, para lo que se podrían analizar algunas formas especiales de árboles de evaluación que se estudian en el análisis de complejidad [45].
- Como queda dicho en el Capítulo 12, es fácil adaptar nuestras técnicas para inferir LBFs en el coste neto de programas terminantes. Una dirección futura sería resolver este programa para programas no terminantes.
- Las técnicas del Capítulo 12 requieren que el usuario proporcione algunas UBF plantilla. Se podría investigar cómo generar esas plantillas automáticamente, sobre la sintaxis del CRS.
- Los métodos de QE usados para las contribuciones del Capítulo 13 pueden aplicarse a los otros métodos de resolución de CRSs. Un ejemplo es para inferir qué expresiones lineales crecen o decrecen de manera aritmética o geométrica, lo cual se necesita para transformar los CRSs en RRs [14], Otros ejemplos son para construir un procedimiento alternativo de maximización, o para inferir funciones de rango no lineales.

Estas mejoras se dirigen a mejorar la precisión y la aplicabilidad del análisis de coste.

Bibliography

[1] ELVIRA ALBERT, DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, JESÚS CORREAS, ANTONIO FLORES-MONTOYA, SAMIR GENAIM, MIGUEL GÓMEZ-ZAMALLOA, ABU NASER MASUD, GERMAN PUEBLA, JOSÉ MIGUEL ROJAS, GUILLERMO ROMÁN-DÍEZ, AND DAMIANO ZANARDINI. Automatic Inference of Bounds on Resource Consumption. In Elena Giachino, Reiner Hähnle, Frank S. de Boer, and Marcello M. Bonsangue, editors of the Proceedings of *International Symposium Formal Methods for Components and Objects (FMCO)*, volume 7866 of *Lecture Notes in Computer Science*, pages 119–144. Springer, September 2013.

[2] ELVIRA ALBERT, DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, SAMIR GENAIM, AND GERMAN PUEBLA. Asymptotic Resource Usage Bounds. In Zhenjiang Hu, editor of the Proceedings of *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 5904 of *Lecture Notes in Computer Science*, pages 294–310. Springer, December 2009.

[3] ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, MIGUEL GÓMEZ-ZAMALLOA, AND GERMÁN PUEBLA. COSTABS: a cost and termination analyzer for ABS. In Oleg Kiselyov and Simon Thompson, editors of the Proceedings of *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 151–154. ACM, January 2012.

[4] ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, ISRAEL HERRAIZ, AND GERMAN PUEBLA. Comparing Cost Functions in Resource Analysis. In Marko C. J. D. van Eekelen and Olha Shkaravska, editors of the Proceedings of *International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA)*, volume 6324 of *Lecture Notes in Computer Science*, pages 1–17. Springer, November 2010.

[5] ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, AND GERMÁN PUEBLA. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

[6] ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, GERMÁN PUEBLA, AND GUILLERMO ROMÁN-DÍEZ. Conditional Termination of Loops over Heap-allocated Data. *Science of Computer Programming*, 2014. In press.

[7] ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, GERMAN PUEBLA, AND DAMIANO ZANARDINI. COSTA: Design and Implementation of a Cost and

Termination Analyzer for Java Bytecode. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors of the Proceedings of *International Symposium Formal Methods for Components and Objects (FMCO)*, volume 5382 of *Lecture Notes in Computer Science*, pages 113–132. Springer, October 2008.

[8] ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, GERMÁN PUEBLA, AND DAMIANO ZANARDINI. Removing useless variables in cost analysis of Java bytecode. In Roger L. Wainwright and Hisham Haddad, editors of the Proceedings of *ACM Symposium on Applied computing (SAC)*, pages 368–375. ACM, March 2008.

[9] ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, GERMÁN PUEBLA, AND DAMIANO ZANARDINI. Resource Usage Analysis and Its Application to Resource Certification. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors of the Proceedings of *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 258–288. Springer, 2009.

[10] ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, GERMAN PUEBLA, AND DAMIANO ZANARDINI. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

[11] ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, AND DAMIANO ZANARDINI. Task-level analysis for a language with async/finish parallelism. In Jan Vitek and Bjorn De Sutter, editors of the Proceedings of *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 21–30. ACM, 2011.

[12] ELVIRA ALBERT, JESÚS CORREAS, GERMÁN PUEBLA, AND GUILLERMO ROMÁN-DÍEZ. Quantified Abstractions of Distributed Systems. In Einar Broch Johnsen and Luigia Petre, editors of the Proceedings of *Integrated Formal Methods (iFM)*, volume 7940 of *Lecture Notes in Computer Science*, pages 285–300. Springer, 2013.

[13] ELVIRA ALBERT, SAMIR GENAIM, AND MIGUEL GÓMEZ-ZAMALLOA. Heap space analysis for garbage collected languages. *Science of Computer Programming*, 78(9):1427–1448, 2013.

[14] ELVIRA ALBERT, SAMIR GENAIM, AND ABU NASER MASUD. On the Inference of Resource Usage Upper and Lower Bounds. *ACM Transactions on Computational Logic*, 14(3):22:1–22:35, August 2013.

[15] CHRISTOPHE ALIAS, ALAIN DARTE, PAUL FEAUTRIER, AND LAURE GONNORD. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In Radhia Cousot and Matthieu Martel, editors of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 6337 of *Lecture Notes in Computer Science*, pages 117–133. Springer, September 2010.

[16] DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Handling Non-linear Operations in the Value Analysis of COSTA. *Electronic Notes in Theoretical Computer Science*, 279(1):3–17, 2011. Proceedings of Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE).

[17] DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Precise Cost Analysis via Local Reasoning. In Dang Van Hung and Mizuhito Ogawa, editors of the Proceedings of *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 8172 of *Lecture Notes in Computer Science*, pages 319–333. Springer, October 2013.

[18] DIEGO ESTEBAN ALONSO-BLAS AND SAMIR GENAIM. On the Limits of the Classical Approach to Cost Analysis. In Antoine Miné and David Schmidt, editors of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 7460 of *Lecture Notes in Computer Science*, pages 405–421. Springer, September 2012.

[19] HUGH ANDERSON, SIAU-CHENG KHOO, STEFAN ANDREI, AND BEATRICE LUCA. Calculating Polynomial Runtime Properties. In Kwangkeun Yi, editor of the Proceedings of *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *Lecture Notes in Computer Science*, pages 230–246. Springer, November 2005.

[20] JASON ANSEL AND CY P. CHAN. PetaBricks. *Crossroads*, 17(1):32–37, 2010.

[21] ANDREW W. APPEL. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[22] MICHAEL ARMBRUST, ARMANDO FOX, REAN GRIFFITH, ANTHONY D. JOSEPH, RANDY KATZ, ANDY KONWINSKI, GUNHO LEE, DAVID PATTERSON, ARIEL RABKIN, ION STOICA, AND MATEI ZAHARIA. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, April 2010.

[23] ROBERT ATKEY. Amortised Resource Analysis with Separation Logic. *Logical Methods in Computer Science*, 7(2), 2011.

[24] ROBERTO BAGNARA, PATRICIA M. HILL, AND ENEA ZAFFANELLA. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1-2):3–21, 2008.

[25] ROBERTO BAGNARA, PATRICIA M. HILL, AND ENEA ZAFFANELLA. Weakly-relational shapes for numeric abstractions: improved algorithms and proofs of correctness. *Formal Methods in System Design*, 35(3):279–323, 2009.

[26] ROBERTO BAGNARA, FRED MESNARD, ANDREA PESCETTI, AND ENEA ZAFFANELLA. A new look at the automatic synthesis of linear ranking functions. *Information and Computation*, 215:47–67, 2012.

[27] ROBERTO BAGNARA, ANDREA PESCETTI, ALESSANDRO ZACCAGNINI, AND ENEA ZAFFANELLA. PURRS: Towards Computer Algebra Support for Fully Automatic Worst-Case Complexity Analysis. *CoRR*, abs/cs/0512056, 2005.

[28] ROBERTO BAGNARA, ENRIC RODRÍGUEZ-CARBONELL, AND ENEA ZAFFANELLA. Generation of Basic Semi-algebraic Invariants Using Convex Polyhedra. In Chris Hankin and Igor Siveroni, editors of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 3672 of *Lecture Notes in Computer Science*, pages 19–34. Springer, September 2005.

[29] CLARK BARRETT, AARON STUMP, AND CESARE TINELLI. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors of the Proceedings of *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[30] FLORENCE BENOY AND ANDY KING. Inferring Argument Size Relationships with CLP(R). In John P. Gallagher, editor of the Proceedings of *International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)*, volume 1207 of *Lecture Notes in Computer Science*, pages 204–223. Springer, August 1997.

[31] RALPH BENZINGER. Automated complexity analysis of Nuprl extracted programs. *Journal of Functional Programming*, 11(1):3–31, 2001.

[32] RALPH BENZINGER. Automated higher-order complexity analysis. *Theoretical Computer Science*, 318(1-2):79–103, 2004.

[33] DOMINIQUE BOUCHER AND MARC FEELEY. Abstract Compilation: A New Implementation Paradigm for Static Analysis. In Tibor Gyimóthy, editor

of the Proceedings of *International Conference on Compiler Construction (CC)*, volume 1060 of *Lecture Notes in Computer Science*, pages 192–207. Springer, April 1996.

[34] AARON R. BRADLEY AND ZOHAR MANNA. *The Calculus of Computation - Decision Procedures with Applications to Verification.* Springer, 2007.

[35] BRIAN CAMPBELL. Amortised Memory Analysis Using the Depth of Data Structures. In Giuseppe Castagna, editor of the Proceedings of *European Symposium on Programming (ESOP)*, volume 5502 of *Lecture Notes in Computer Science*, pages 190–204. Springer, March 2009.

[36] YONGJAE CHA, MARK VAN HOEIJ, AND GILES LEVY. Solving recurrence relations using local invariants. In Wolfram Koepf, editor of the Proceedings of *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 303–309. ACM, July 2010.

[37] LIQIAN CHEN, ANTOINE MINÉ, JI WANG, AND PATRICK COUSOT. Linear Absolute Value Relation Analysis. In Gilles Barthe, editor of the Proceedings of *European Symposium on Programming (ESOP)*, volume 6602 of *Lecture Notes in Computer Science*, pages 156–175. Springer, March 2011.

[38] YINGHUA CHEN, BICAN XIA, LU YANG, AND NAIJUN ZHAN. Generating Polynomial Invariants with DISCOVERER and QEPCAD. In Cliff B. Jones, Zhiming Liu, and Jim Woodcock, editors of the Proceedings of *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *Lecture Notes in Computer Science*, pages 67–82. Springer, September 2007.

[39] THOMAS CLUZEAU AND MARK VAN HOEIJ. Computing Hypergeometric Solutions of Linear Recurrence Equations. *Appl. Algebra Eng. Commun. Comput.*, 17(2):83–115, 2006.

[40] MICHAEL CODISH. Efficient Goal Directed Bottom-Up Evaluation of Logic Programs. *Journal of Logic Programming*, 38(3):355–370, 1999.

[41] JACQUES COHEN. Computer-Assisted Microanalysis of Programs. *Communications of the ACM*, 25(10):724–733, 1982.

[42] JACQUES COHEN AND JOEL KATCOFF. Symbolic Solution of Finite-Difference Equations. *ACM Transactions on Mathematical Software*, 3(3):261–271, 1977.

[43] JACQUES COHEN AND ALINE WEITZMAN. Software Tools for Micro-analysis of Programs. *Software - Practice and Experience*, 22(9):777–808, 1992.

[44] JACQUES COHEN AND CARL ZUCKERMAN. Two Languages for Estimating Program Efficiency. *Communications of the ACM*, 17(6):301–308, 1974.

[45] THOMAS H. CORMEN, CHARLES E. LEISERSON, RONALD L. RIVEST, AND CLIFFORD STEIN. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[46] PATRICK COUSOT. Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes. Thèse d'État ès sciences mathématiques, Université Joseph Fourier, Grenoble, France, March 1978.

[47] PATRICK COUSOT. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In Radhia Cousot, editor of the Proceedings of *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3385 of *Lecture Notes in Computer Science*, pages 1–24. Springer, June 2005.

[48] PATRICK COUSOT. Topics in Abstract Interpretation. `http://www.di.ens.fr/~cousot/AI/`, August 2008.

[49] PATRICK COUSOT, RADHIA COUSOT, JÉRÔME FERET, LAURENT MAUBORGNE, ANTOINE MINÉ, AND XAVIER RIVAL. Why does Astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.

[50] KARL CRARY AND STEPHANIE WEIRICH. Resource Bound Certification. In Mark N. Wegman and Thomas W. Reps, editors of the Proceedings of *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 184–198. ACM, January 2000.

[51] NORMAN DANNER, JENNIFER PAYKIN, AND JAMES S. ROYER. A static cost analysis for a higher-order language. In Matthew Might, David Van Horn, Andreas Abel, and Tim Sheard, editors of the Proceedings of *ACM SIGPLAN Workshop on Programming Languages meets Program Verification (PLPV)*, pages 25–34. ACM, January 2013.

[52] JAVIER DE DIOS AND RICARDO PEÑA. Certification of Safe Polynomial Memory Bounds. In Michael Butler and Wolfram Schulte, editors of the Proceedings of *International Symposium on Formal Methods (FM)*, volume 6664 of *Lecture Notes in Computer Science*, pages 184–199. Springer, June 2011.

[53] LEONARDO MENDONÇA DE MOURA AND NIKOLAJ BJØRNER. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors of the Proceedings of *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, April 2008.

[54] SAUMYA K. DEBRAY AND NAI-WEI LIN. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, 1993.

[55] XIAO YAN DENG, GREG MICHAELSON, AND PHILIP W. TRINDER. Cost-driven autonomous mobility. *Computer Languages, Systems & Structures*, 36(1):34–59, 2010.

[56] ANDREAS DOLZMANN, ANDREAS SEIDL, AND THOMAS STURM. *Redlog User Manual, Edition 3.1*, November 2006.

[57] PHILIPPE FLAJOLET, BRUNO SALVY, AND PAUL ZIMMERMANN. Automatic Average-Case Analysis of Algorithm. *Theoretical Computer Science*, 79(1):37–109, 1991.

[58] PHILIPPE FLAJOLET AND ROBERT SEDGEWICK. *Analytic Combinatorics*. Cambridge University Press, 2009.

[59] JÜRGEN GIESL, THOMAS STRÖDER, PETER SCHNEIDER-KAMP, FABIAN EMMES, AND CARSTEN FUHS. Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs. In Danny De Schreye, Gerda Janssens, and Andy King, editors of the Proceedings of *International Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 1–12. ACM, September 2012.

[60] JÜRGEN GIESL, RENÉ THIEMANN, PETER SCHNEIDER-KAMP, AND STEPHAN FALKE. Automated Termination Proofs with AProVE. In Vincent van Oostrom, editor of the Proceedings of *International Conference on Rewriting Techniques and Applications (RTA)*, volume 3091 of *Lecture Notes in Computer Science*, pages 210–220. Springer, June 2004.

[61] DANIEL H. GREENE AND DONALD E. KNUTH. *Mathematics for the Analysis of Algorithms*. Birkhäuser, 2008.

[62] BERND GROBAUER. Cost Recurrences for DML Programs. In Benjamin C. Pierce, editor of the Proceedings of *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 253–264. ACM, September 2001.

[63] BHARGAV S. GULAVANI AND SUMIT GULWANI. A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In Aarti Gupta and Sharad Malik, editors of the Proceedings of *International Conference on Computer Aided Verification (CAV)*, volume 5123 of *Lecture Notes in Computer Science*, pages 370–384. Springer, July 2008.

[64] SUMIT GULWANI, SAGAR JAIN, AND ERIC KOSKINEN. Control-flow refinement and progress invariants for bound analysis. In Michael Hind and Amer Diwan, editors of the Proceedings of *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 375–385. ACM, June 2009.

[65] SUMIT GULWANI, KRISHNA K. MEHRA, AND TRISHUL M. CHILIMBI. SPEED: precise and efficient static estimation of program computational complexity. In Zhong Shao and Benjamin C. Pierce, editors of the Proceedings of *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 127–139. ACM, January 2009.

[66] SUMIT GULWANI AND FLORIAN ZULEGER. The reachability-bound problem. In Benjamin G. Zorn and Alexander Aiken, editors of the Proceedings of *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 292–304. ACM, June 2010.

[67] NICOLAS HALBWACHS. Détermination automatique de relations linéaires vérifiées par les variables d'un programme. Ph.D. Thesis, Institut National Polytechnique de Grenoble, March 1978.

[68] ANTHONY C. HEARN. REDUCE *User's Manual, Version 3.8*, February 2004.

[69] MANUEL V. HERMENEGILDO, FRANCISCO BUENO, MANUEL CARRO, PEDRO LÓPEZ-GARCÍA, EDISON MERA, JOSÉ F. MORALES, AND GERMÁN PUEBLA. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming*, 12(1-2):219–252, 2012.

[70] MANUEL V. HERMENEGILDO, RICHARD WARREN, AND SAUMYA K. DEBRAY. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–366, 1992.

[71] TIMOTHY J. HICKEY AND JACQUES COHEN. Automating program analysis. *Journal of the ACM*, 35(1):185–220, 1988.

[72] C. A. R. HOARE. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[73] JAN HOFFMANN, KLAUS AEHLIG, AND MARTIN HOFMANN. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems*, 34(3):14, 2012.

[74] JAN HOFFMANN AND ZHONG SHAO. Type-Based Amortized Resource Analysis with Integers and Arrays. In Michael Codish and Eijiro Sumii, editors of the Proceedings of *International Symposium on Functional and Logic Programming (FLOPS)*, Lecture Notes in Computer Science. Springer, 2014. In press.

[75] MARTIN HOFMANN AND STEFFEN JOST. Static prediction of heap space usage for first-order functional programs. In Alex Aiken and Greg Morrisett, editors of the Proceedings of *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 185–197. ACM, 2003.

[76] MARTIN HOFMANN AND DULMA RODRIGUEZ. Automatic Type Inference for Amortised Heap-Space Analysis. In Matthias Felleisen and Philippa Gardner, editors of the Proceedings of *European Symposium on Programming (ESOP)*, volume 7792 of *Lecture Notes in Computer Science*, pages 593–613. Springer, March 2013.

[77] RODNEY R. HOWELL. On Asymptotic Notations with Multiple Variables. Technical Report 2007-4, Department of Computing and Information Sciences, Kansas State University, Manhatan, Kansas, January 2008.

[78] JOHN IVIE. Some MACSYMA Programs for Solving Recurrence Relations. *ACM Transactions on Mathematical Software*, 4(1):24–33, 1978.

[79] STEFFEN JOST. Automated Amortised Analysis. Ph.D. Thesis, Ludwig Maximillians Universitat Munich, August 2010.

[80] STEFFEN JOST, KEVIN HAMMOND, HANS-WOLFGANG LOIDL, AND MARTIN HOFMANN. Static determination of quantitative resource usage for higher-order programs. In Manuel V. Hermenegildo and Jens Palsberg, editors of the Proceedings of *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 223–236. ACM, January 2010.

[81] DEEPAK KAPUR. Automatically Generating Loop Invariants Using Quantifier Elimination. In Franz Baader, Peter Baumgartner, Robert Nieuwenhuis, and Andrei Voronkov, editors of the Proceedings of *Deduction*

and Applications, volume 05431 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, October 2006.

[82] RICHARD M. KARP. Probabilistic Recurrence Relations. *Journal of the ACM*, 41(6):1136–1150, 1994.

[83] RICHARD M. KARP, RAYMOND E. MILLER, AND SHMUEL WINOGRAD. The Organization of Computations for Uniform Recurrence Equations. *Journal of the ACM*, 14(3):563–590, 1967.

[84] TAKUMI KASAI AND AKEO ADACHI. A Characterization of Time Complexity by Simple Loop Programs. *Journal of Computer and System Sciences (JCSS)*, 20(1):1–17, 1980.

[85] OWEN KASER, C. R. RAMAKRISHNAN, AND SHAUNAK PAWAGI. On the Conversion of Indirect to Direct Recursion. *ACM Letters on Programming Languages and Systems*, 2(1-4):151–164, 1993.

[86] STEVE KERRISON, UMER LIQAT, KYRIAKOS GEORGIOU, ALEJANDRO SERRANO MENA, NEVILLE GRECH, PEDRO LÓPEZ-GARCÍA, KERSTIN EDER, AND MANUEL V. HERMENEGILDO. Energy Consumption Analysis of Programs based on XMOS ISA-Level Models. In Gopal Gupta and Ricardo Peña, editors of the Proceedings of *International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)*, September 2013.

[87] DONALD E. KNUTH. Structured Programming with go to Statements. *ACM Computing Surveys*, 6(4):261–301, 1974.

[88] DONALD E. KNUTH. Big Omicron and Big Omega and Big Theta. *SIGACT News*, 8(2):18–24, April 1976.

[89] DANIEL KROENING AND OFER STRICHMAN. *Decision Procedures: An Algorithmic Point of View*. Springer, 1 edition, 2008.

[90] UGO DAL LAGO AND BARBARA PETIT. The geometry of types. In Roberto Giacobazzi and Radhia Cousot, editors of the Proceedings of *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 167–178. ACM, January 2013.

[91] DAVID LESENS. Using Static Analysis in Space: Why Doing so? In Radhia Cousot and Matthieu Martel, editors of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 6337 of *Lecture Notes in Computer Science*, pages 51–70. Springer, September 2010.

[92] SENLIN LIANG. Non-termination Analysis and Cost-Based Query Optimization of Logic Programs. In Markus Krötzsch and Umberto Straccia, editors of the Proceedings of *International Conference on Web Reasoning and Rule Systems (RR)*, volume 7497 of *Lecture Notes in Computer Science*, pages 284–290. Springer, September 2012.

[93] TIM LINDHOLM, FRANK YELLIN, GILAD BRACHA, AND ALEX BUCKLEY. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley Professional, 2013.

[94] PEDRO LÓPEZ-GARCÍA, LUTHFI DARMAWAN, AND FRANCISCO BUENO. A Framework for Verification and Debugging of Resource Usage Properties: Resource Usage Verification. In Manuel V. Hermenegildo and Torsten Schaub, editors of the Proceedings of *International Conference on Logic Programming (ICLP)(Technical Communications)*, volume 7 of *LIPIcs*, pages 104–113. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, July 2010.

[95] PEDRO LÓPEZ-GARCÍA, MANUEL V. HERMENEGILDO, AND SAUMYA K. DEBRAY. A Methodology for Granularity-Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation*, 21(4):715–734, 1996.

[96] DANIEL LE MÉTAYER. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, 1988.

[97] MANUEL MONTENEGRO, OLHA SHKARAVSKA, MARKO C. J. D. VAN EEKELEN, AND RICARDO PEÑA. Interpolation-Based Height Analysis for Improving a Recurrence Solver. In Ricardo Peña, Marko C. J. D. van Eekelen, and Olha Shkaravska, editors of the Proceedings of *International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA)*, volume 7177 of *Lecture Notes in Computer Science*, pages 36–53. Springer, 2012.

[98] JORGE A. NAVAS, MARIO MÉNDEZ-LOJO, AND MANUEL V. HERMENEGILDO. User-Definable Resource Usage Bounds Analysis for Java Bytecode. *Electronic Notes in Theoretical Computer Science*, 253(5):65–82, 2009. Proceedings of Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE).

[99] JORGE A. NAVAS, EDISON MERA, PEDRO LÓPEZ-GARCÍA, AND MANUEL V. HERMENEGILDO. User-Definable Resource Bounds Analysis for Logic Programs. In Verónica Dahl and Ilkka Niemelä, editors of the Proceedings

of *International Conference on Logic Programming (ICLP)*, volume 4670 of *Lecture Notes in Computer Science*, pages 348–363. Springer, September 2007.

[100] FLEMMING NIELSON, HANNE RIIS NIELSON, AND CHRIS HANKIN. *Principles of Program Analysis*. Springer, 2005.

[101] FLEMMING NIELSON, HANNE RIIS NIELSON, AND HELMUT SEIDL. Automatic Complexity Analysis. In Daniel Le Métayer, editor of the Proceedings of *European Symposium on Programming (ESOP)*, volume 2305 of *Lecture Notes in Computer Science*, pages 243–261. Springer, April 2002.

[102] LARS NOSCHINSKI, FABIAN EMMES, AND JÜRGEN GIESL. Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs. *Journal of Automated Reasoning*, 51(1):27–56, 2013.

[103] CHRIS OKASAKI. *Purely functional data structures*. Cambridge University Press, 1999.

[104] MARKO PETKOVSEK. Hypergeometric Solutions of Linear Recurrences with Polynomial Coefficents. *Journal of Symbolic Computation*, 14(2/3):243–264, 1992.

[105] MARKO PETKOVSEK, HERBERT WILF, AND DORON ZEILBERGER. *A=B*. A K Peters Ltd., 1996.

[106] LYLE HAROLD RAMSHAW. Formalizing the analysis of algorithms. Ph.D. Thesis, Stanford University, Stanford, CA, USA, 1979.

[107] JOHN C. REYNOLDS. An Overview of Separation Logic. In Bertrand Meyer and Jim Woodcock, editors of the Proceedings of *Verified Software: Theories, Tools, and Experiments*, volume 4171 of *Lecture Notes in Computer Science*, pages 460–469. Springer, October 2005.

[108] XAVIER RIVAL AND LAURENT MAUBORGNE. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems*, 29(5), 2007.

[109] DULMA RODRIGUEZ. Amortised Resource Analysis for Object-Oriented Programs. Ph.D. Thesis, Ludwig Maximillians Universitat Munich, October 2012.

[110] ENRIC RODRÍGUEZ-CARBONELL AND DEEPAK KAPUR. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming*, 64(1):54–75, 2007.

[111] OREN PATASHNIK RONALD L. GRAHAM, DONALD E. KNUTH. *Concrete Mathematics*. Addison-Weasley, 2nd edition, 1994.

[112] MADS ROSENDAHL. Automatic Complexity Analysis. In Proceedings of *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 144–156, 1989.

[113] BRUNO SALVY AND PAUL ZIMMERMANN. GFUN: a Maple package for the manipulation of generating and holonomic functions in one variable. *ACM Transactions on Mathematical Software*, 20(2):163–177, 1994.

[114] SRIRAM SANKARANARAYANAN, FRANJO IVANCIC, ILYA SHLYAKHTER, AND AARTI GUPTA. Static Analysis in Disjunctive Numerical Domains. In Kwangkeun Yi, editor of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 4134 of *Lecture Notes in Computer Science*, pages 3–17. Springer, August 2006.

[115] SRIRAM SANKARANARAYANAN, HENNY SIPMA, AND ZOHAR MANNA. Non-linear loop invariant generation using Gröbner bases. In Neil D. Jones and Xavier Leroy, editors of the Proceedings of *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 318–329. ACM, January 2004.

[116] SRIRAM SANKARANARAYANAN, HENNY B. SIPMA, AND ZOHAR MANNA. Constructing invariants for hybrid systems. *Formal Methods in System Design*, 32(1):25–55, 2008.

[117] GABRIEL SCHERER AND JAN HOFFMANN. Tracking Data-Flow with Open Closure Types. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors of the Proceedings of *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 8312 of *Lecture Notes in Computer Science*, pages 710–726. Springer, December 2013.

[118] ROBERT SEDGEWICK AND PHILIPPE FLAJOLET. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, 2nd edition, January 2013.

[119] ALEJANDRO SERRANO, PEDRO LÓPEZ-GARCÍA, FRANCISCO BUENO, AND MANUEL V. HERMENEGILDO. Sized Type Analysis for Logic Programs. *Theory and Practice of Logic Programming*, 13(4-5-Online-Supplement), 2013.

[120] OLHA SHKARAVSKA AND MARKO C. J. D. VAN EEKELEN. Univariate polynomial solutions of algebraic difference equations. *Journal of Symbolic Computation*, 60:15–28, 2014.

[121] OLHA SHKARAVSKA, MARKO C. J. D. VAN EEKELEN, AND RON VAN KESTEREN. Polynomial Size Analysis of First-Order Shapely Functions. *Logical Methods in Computer Science*, 5(2), 2009.

[122] HUGO R. SIMÕES, PEDRO B. VASCONCELOS, MÁRIO FLORIDO, STEFFEN JOST, AND KEVIN HAMMOND. Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In Peter Thiemann and Robby Bruce Findler, editors of the Proceedings of *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 165–176. ACM, September 2012.

[123] FAUSTO SPOTO, FRED MESNARD, AND ÉTIENNE PAYET. A termination analyzer for Java bytecode based on path-length. *ACM Transactions on Programming Languages and Systems*, 32(3), 2010.

[124] SAURABH SRIVASTAVA, SUMIT GULWANI, AND JEFFREY S. FOSTER. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):497–518, 2013.

[125] BJARNE STEENSGAARD. Points-to Analysis in Almost Linear Time. In Hans-Juergen Boehm and Guy L. Steele Jr., editors of the Proceedings of *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 32–41. ACM, January 1996.

[126] DAVID R. STOUTEMYER. Automatic Asymptotic and Big-O Calculations Via Computer Algebra. *SIAM Journal on Computing*, 8(3):287–299, 1979.

[127] THOMAS STURM AND ASHISH TIWARI. Verification and synthesis using real quantifier elimination. In Éric Schost and Ioannis Z. Emiris, editors of the Proceedings of *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 329–336. ACM, June 2011.

[128] ANKUR TALY, SUMIT GULWANI, AND ASHISH TIWARI. Synthesizing switching logic using constraint solving. *International Journal on Software Tools for Technology Transfer*, 13(6):519–535, 2011.

[129] ROBERT ENDRE TARJAN. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[130] PEDRO VASCONCELOS. Space Cost Analysis using Sized Types. Ph.D. Thesis, University of St Andrews, August 2008.

[131] PEDRO B. VASCONCELOS AND KEVIN HAMMOND. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In Philip W. Trinder, Greg Michaelson, and Ricardo Peña, editors of the Proceedings of *Implementation of Functional Languages (IFL)*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer, September 2004.

[132] PHILIP WADLER. Strictness Analysis Aids Time Analysis. In Jeanne Ferrante and P. Mager, editors of the Proceedings of *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 119–132. ACM, January 1988.

[133] BEN WEGBREIT. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, 1975.

[134] BEN WEGBREIT. Verifying Program Performance. *Journal of the ACM*, 23(4):691–699, 1976.

[135] BEN WEGBREIT. Constructive Methods in Program Verification. *IEEE Transactions on Software Engineering (TSE)*, 3(3):193–209, 1977.

[136] TAO WEI, JIAN MAO, WEI ZOU, AND YU CHEN. A New Algorithm for Identifying Loops in Decompilation. In Hanne Riis Nielson and Gilberto Filé, editors of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 4634 of *Lecture Notes in Computer Science*, pages 170–183. Springer, August 2007.

[137] REINHARD WILHELM, JAKOB ENGBLOM, ANDREAS ERMEDAHL, NIKLAS HOLSTI, STEPHAN THESING, DAVID B. WHALLEY, GUILLEM BERNAT, CHRISTIAN FERDINAND, REINHOLD HECKMANN, TULIKA MITRA, FRANK MUELLER, ISABELLE PUAUT, PETER P. PUSCHNER, JAN STASCHULAT, AND PER STENSTRÖM. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions in Embedded Computing Systems*, 7(3), 2008.

[138] PETER Y. H. WONG, ELVIRA ALBERT, RADU MUSCHEVICI, JOSÉ PROENÇA, JAN SCHÄFER, AND RUDOLF SCHLATTE. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *International Journal on Software Tools for Technology Transfer*, 14(5):567–588, 2012.

[139] TING YU AND OWEN KASER. A Note on "On the Conversion of Indirect to Direct Recursion". *ACM Transactions on Programming Languages and Systems*, 19(6):1085–1087, 1997.

[140] FLORIAN ZULEGER, SUMIT GULWANI, MORITZ SINN, AND HELMUT VEITH. Bound Analysis of Imperative Programs with the Size-Change Abstraction. In Eran Yahav, editor of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 6887 of *Lecture Notes in Computer Science*, pages 280–297. Springer, September 2011.

Part III

Supporting Publications of this Thesis

Asymptotic Resource Usage Bounds

Elvira Albert[1], Diego Alonso[1], Puri Arenas[1], Samir Genaim[1], and German Puebla[2]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

Abstract. When describing the resource usage of a program, it is usual to talk in *asymptotic* terms, such as the well-known "big O" notation, whereby we focus on the behaviour of the program for large input data and make a rough approximation by considering as equivalent programs whose resource usage grows at the same rate. Motivated by the existence of *non-asymptotic* resource usage analyzers, in this paper, we develop a novel transformation from a non-asymptotic cost function (which can be produced by multiple resource analyzers) into its asymptotic form. Our transformation aims at producing tight asymptotic forms which do not contain *redundant* subexpressions (i.e., expressions asymptotically subsumed by others). Interestingly, we integrate our transformation at the heart of a cost analyzer to generate asymptotic *upper bounds* without having to first compute their non-asymptotic counterparts. Our experimental results show that, while non-asymptotic cost functions become very complex, their asymptotic forms are much more compact and manageable. This is essential to improve scalability and to enable the application of cost analysis in resource-aware verification/certification.

1 Introduction

A fundamental characteristics of a program is the amount of resources that its execution will require, i.e., its *resource usage*. Typical examples of resources include execution time, memory watermark, amount of data transmitted over the net, etc. *Resource usage analysis* [15,14,8,2,9] aims at automatically estimating the resource usage of programs. Static resource analyzers often produce *cost bound functions*, which have as input the size of the input arguments and return bounds on the resource usage (or *cost*) of running the program on such input.

A well-known mechanism for keeping the size of cost functions manageable and, thus, facilitate human manipulation and comparison of cost functions is *asymptotic analysis*, whereby we focus on the behaviour of functions for large input data and make a rough approximation by considering as equivalent functions which grow at the same rate w.r.t. the size of the input date. The asymptotic point of view is basic in computer science, where the question is typically how to describe the resource implication of scaling-up the size of a computational problem, beyond the "toy" level. For instance, the big O notation is used to define *asymptotic upper bounds*, i.e, given two functions f and g which map natural numbers to real numbers, one writes $f \in O(g)$ to express the fact that there is a natural constant $m \geq 1$

Z. Hu (Ed.): APLAS 2009, LNCS 5904, pp. 294–310, 2009.

and a real constant $c > 0$ s.t. for any $n \geq m$ we have that $f(n) \leq c * g(n)$. Other types of (asymptotic) computational complexity estimates are lower bounds ("Big Omega" notation) and asymptotically tight estimates, when the asymptotic upper and lower bounds coincide (written using "Big Theta"). The aim of *asymptotic resource usage analysis* is to obtain a cost function f_a which is *syntactically simple* s.t. $f_n \in O(f_a)$ (correctness) and ideally also that $f_a \in \Theta(f_n)$ (accuracy), where f_n is the non-asymptotic cost function.

The scopes of non-asymptotic and asymptotic analysis are complementary. Non-asymptotic bounds are required for the estimation of precise execution time (like in WCET) or to predict accurate memory requirements [4]. The motivations for inferring asymptotic bounds are twofold: (1) They are essential during program development, when the programmer tries to reason about the efficiency of a program, especially when comparing alternative implementations for a given functionality. (2) Non-asymptotic bounds can become unmanageably large expressions, imposing huge memory requirements. We will show that asymptotic bounds are syntactically much simpler, can be produced at a smaller cost, and, interestingly, in cases where their non-asymptotic forms cannot be computed.

The main techniques presented in this paper are applicable to obtain asymptotic versions of the cost functions produced by any cost analysis, including lower, upper and average cost analyses. Besides, we will also study how to perform a tighter integration with an upper bound solver which follows the classical approach to static cost analysis by Wegbreit [15]. In this approach, the analysis is parametric w.r.t. a *cost model*, which is just a description of the resources whose usage we should measure, e.g., time, memory, calls to a specific function, etc. and analysis consists of two phases. (1) First, given a program and a cost model, the analysis produces *cost relations* (CRs for short), i.e., a system of recursive equations which capture the resource usage of the program for the given cost model in terms of the sizes of its input data. (2) In a second step, *closed-form*, i.e., non-recursive, upper bounds are inferred for the CRs. How the first phase is performed is heavily determined by the programming language under study and nowadays there exist analyses for a relatively wide range of languages (see, e.g., [2,8,14] and their references). Importantly, such first phase remains the same for both asymptotic and non-asymptotic analyses and thus we will not describe it. The second phase is language-independent, i.e., once the CRs are produced, the same techniques can be used to transform them to closed-form upper bounds, regardless of the programming language used in the first phase. The important point is that this second phase can be modified in order to produce asymptotic upper bounds directly. Our main contributions can be summarized as follows:

1. We adapt the notion of *asymptotic complexity* to cover the analysis of realistic programs whose limiting behaviour is determined by the limiting behaviour of its loops.
2. We present a novel transformation from *non-asymptotic cost functions* into asymptotic form. After some syntactic simplifications, our transformation detects and eliminates subterms which are *asymptotically subsumed* by others while preserving the complexity order.

3. In order to achieve motivation (2), we need to integrate the above transformation within the process of obtaining the cost functions. We present a tight integration into (the second phase of) a resource usage analyzer to generate directly asymptotic upper bounds without having to first compute their non-asymptotic counterparts.
4. We report on a prototype implementation within the COSTA system [3] which shows that we are able to achieve motivations (1) and (2) in practice.

2 Background: Non-asymptotic Upper Bounds

In this section, we recall some preliminary definitions and briefly describe the method of [1] for converting *cost relations* (CRs) into upper bounds in *closed-form*, i.e., without recurrences.

2.1 Cost Relations

Let us introduce some notation. The sets of natural, integer, real, non-zero natural and non-negative real values are denoted respectively by $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{N}^+$ and $\mathbb{R}^+$. We write x, y, and z, to denote variables which range over $\mathbb{Z}$. A *linear expression* has the form $v_0 + v_1x_1 + \ldots + v_nx_n$, where $v_i \in \mathbb{Z}$, $0 \leq i \leq n$. Similarly, a *linear constraint* (over $\mathbb{Z}$) has the form $l_1 \leq l_2$, where l_1 and l_2 are linear expressions. For simplicity we write $l_1 = l_2$ instead of $l_1 \leq l_2 \wedge l_2 \leq l_1$, and $l_1 < l_2$ instead of $l_1 + 1 \leq l_2$. The notation $\bar{t}$ stands for a sequence of entities $t_1, \ldots, t_n$, for some $n>0$. We write φ, ϕ or ψ, to denote sets of linear constraints which should be interpreted as the conjunction of each element in the set and $\varphi_1 \models \varphi_2$ to indicate that the linear constraint φ_1 implies the linear constraint φ_2. Now, the basic building blocks of cost relations are the so-called *cost expressions* e which can be generated using this grammar:

$$e ::= r \mid \mathsf{nat}(l) \mid e + e \mid e * e \mid e^r \mid \log(\mathsf{nat}(l)) \mid n^{\mathsf{nat}(l)} \mid \max(S)$$

where $r \in \mathbb{R}^+$, $n \in \mathbb{N}^+$, l is a linear expression, S is a non empty set of cost expressions, $\mathsf{nat} : \mathbb{Z} \to \mathbb{N}$ is defined as $\mathsf{nat}(v) = \max(\{v, 0\})$, and the base of the log is 2 (since any other base can be rewritten to 2). Observe that linear expressions are always wrapped by nat as we explain below.

Example 1. Consider the simple Java method m shown in Fig. 1, which invokes the auxiliary method g, where x is a linear list of boolean values implemented

static void m(List x, int i, int n){	(1) $\langle C_m(i,n) = 3$
while (i<n){	$\quad, \varphi_1 = \{i \geq n\}\rangle$
if (x.data) {g(i,n); i++;}	(2) $\langle C_m(i,n) = 15 + C_g(i,n) + C_m(i',n)$
else {g(0,i); n=n-1;}	$\quad, \varphi_2 = \{i < n, i' = i + 1\}\rangle$
x=x.next;	(3) $\langle C_m(i,n) = 17 + C_g(0,i) + C_m(i,n')$
}}	$\quad, \varphi_3 = \{i < n, , n' = n - 1\}\rangle$

Fig. 1. Java method and CR

in the standard way. For this method, the COSTA analyzer outputs the cost expression C_m^+=6+nat$(n-i)$*max$(\{21+5*\mathsf{nat}(n-1), 19+5*\mathsf{nat}(n-i)\})$ as an upper bound on the number of *bytecode* instructions that m executes. Each Java instruction is compiled to possibly several bytecode instructions, but this is not relevant to this work. We are assuming that an upper bound on the number of executed instructions in g is $C_g^+(a,b)$=4+5*nat$(b-a)$. Observe that the use of nat is required in order to avoid incorrectly evaluating upper bounds to negative values. When $i \geq n$, the cost associated to the recursive cases has to be nulled out, this effect is achieved with nat$(n-i)$ since it will evaluate to 0. □

W.l.o.g., we formalize our mechanism by assuming that all recursions are *direct* (i.e., all cycles are of length one). Direct recursion can be automatically achieved by applying *Partial Evaluation* [11] (see [1] for the technical details).

Definition 1 (Cost Relation). *A* cost relation system $\mathcal{S}$ *is a set of equations of the form* $\langle C(\bar{x}) = e + \sum_{i=1}^{k} D_i(\bar{y}_i), \varphi\rangle$ *with* $k \geq 0$*, where* C *and* D_i *are cost relation symbols, all variables* $\bar{x}$ *and* $\bar{y}_i$ *are distinct variables;* e *is a cost expression; and* φ *is a set of linear constraints over* $\bar{x} \cup vars(e) \bigcup_{i=1}^{k} \bar{y}_i$.

Example 2. The *cost relation* (CR for short) associated to method m is shown in Fig. 1 (right). The relations C_m and C_g capture, respectively, the costs of the methods m and g. Intuitively, in CRs, variables represent the sizes of the corresponding data structures in the program and in the case of integer variables they represent their integer value. Eq. 1 is a base case and captures the case where the loop body is not executed. It can be observed that we have two recursive equations (Eq. 2 and Eq. 3) which capture the respective costs of the then and else branches within the while loop. As the list x has been abstracted to its length, the values of x.data are not visible in the CR and the two equations have the same (incomplete) guard, which results in a non-deterministic CR. Also, variables which do not affect the cost (e.g., x) do not appear in the CR. How to automatically obtain a CR from a program is the subject of the first phase of cost analysis as described in Sec. 1. More details can be found in [2,8,14,15]. □

2.2 Non-asymptotic Upper-Bounds

We now describe the approach of [1] to infer the upper bound of Ex. 1 from the equations in Ex. 2. It starts by computing upper bounds for CRs which do not depend on any other CRs, referred to as *standalone cost relations*, and continues by replacing the computed upper bounds on the equations which call such relations. For instance, after computing the upper bound for g shown in Ex. 1, the cost relation in Ex. 2 becomes standalone:

(1) $\langle C_m(i,n) = 3 \quad , \ \varphi_1 = \{i \geq n\}\rangle$
(2) $\langle C_m(i,n) = 15 + \mathsf{nat}(n-i) + C_m(i',n) \ , \ \varphi_2 = \{i < n, i' = i+1\}\rangle$
(3) $\langle C_m(i,n) = 17 + \mathsf{nat}(i) + C_m(i,n') \ , \ \varphi_3 = \{i < n, n' = n-1\}\rangle$

Given a standalone CR made up of nb base cases of the form $\langle C(\bar{x})=base_j, \varphi_j\rangle$, $1\leq j\leq nb$ and nr recursive equations of the form, $\langle C(\bar{x})=rec_j+\sum_{i=1}^{k_j} C(\bar{y}_i), \varphi_j\rangle$, $1\leq j\leq nr$, an upper bound can be computed as:

$$(*) \quad C(\bar{x})^+ = \mathsf{l_b} * worst(\{base_1, \ldots, base_{nb}\}) + \mathsf{l_r} * worst(\{rec_1, \ldots, rec_{nr}\})$$

where $\mathsf{l_b}$ and $\mathsf{l_r}$ are, respectively, upper bounds of the number of visits to the base cases and recursive equations and $worst(\{Set\})$ denotes the worst-case (the maximum) value that the expressions in Set can take. Below, we describe the method in [1] to approximate the above upper bound.

Bounds on the Number of Application of Equations. The first dimension of the problem is to bound the maximum number of times an equation can be applied. This can be done by examining the structure of the CR (i.e., the number of explicit recursive calls in the equations), together with how the values of the arguments change when calling recursively (i.e., the linear constraints).

We first explain the problem for equations that have at most one recursive call in their bodies. In the above CR, when calling C_m recursively in (2), the first argument i of C_m increases by 1 and in (3) the second argument n decreases by 1. Now suppose that we define a function $f(a, b) = b - a$. Then, we can observe that $\varphi_2 \models f(i, n) > f(i', n) \wedge f(i, n) \geq 0$ and $\varphi_3 \models f(i, n) > f(i, n') \wedge f(i, n) \geq 0$, i.e, for both equations we can guarantee that they will not be applied more than $\mathsf{nat}(f(i_0, n_0)) = \mathsf{nat}(n_0 - i_0)$ times, where i_0 and n_0 are the initial values for the two variables. Functions such as f are usually called *ranking functions* [13]. Given a cost relation $C(\bar{x})$, we denote by $f_C(\bar{x})$ a ranking function for all loops in C. Now, consider that we add an equation that contains two recursive calls:

$$(4)\ \langle C_m(i, n) = C_m(i, n') + C_m(i, n')\ \ ,\ \varphi_4 = \{i < n, n' = n - 1\}\rangle$$

then the recursive equations would be applied in the worst-case $\mathsf{l_r} = 2^{\mathsf{nat}(n-i)} - 1$ times, which in this paper, we simplify to $\mathsf{l_r} = 2^{\mathsf{nat}(n-i)}$ to avoid having negative constants that do not add any technical problem to asymptotic analysis. This is because each call generates 2 recursive calls, and in each call the argument n decreases at least by 1. In addition, unlike the above examples, the base-case equation would be applied in the worst-case an exponential number of times. In general, a CR may include several base-case and recursive equations whose guards, as shown in the example, are not necessarily mutually exclusive, which means that at each evaluation step there are several equations that can be applied. Thus, the worst-case of applications is determined by the fourth equation, which has two recursive calls, while the worst cost of each application will be determined by the first equation, which contributes the largest direct cost. In summary, the bounds on the number of application of equations are computed as follows:

$$\mathsf{l_r} = \begin{cases} \mathsf{nr}^{\mathsf{nat}(f_C(\bar{x}))} & \text{if } \mathsf{nr} > 1 \\ \mathsf{nat}(f_C(\bar{x})) & \text{otherwise} \end{cases} \qquad \mathsf{l_b} = \begin{cases} \mathsf{nr}^{\mathsf{nat}(f_C(\bar{x}))} & \text{if } \mathsf{nr} > 1 \\ 1 & \text{otherwise} \end{cases}$$

where nr is the maximum number of recursive calls which appear in a single equation. A fundamental point to note is that the (linear) combination of

variables which approximates the number of iterations of loops is wrapped by nat. This will influence our definition of asymptotic complexity. In logarithmic cases, we can further refine the ranking function and obtain a tighter upper bound. If each recursive equation satisfies $\varphi_j \models f_C(\bar{x}) \geq k * f_C(\bar{y}_i)$, $1 \leq i \leq nr$, where $k>1$ is a constant, then we can infer that $\mathsf{I_r}$ is bounded by $\lceil \log_k(\mathsf{nat}(f_C(\bar{x}))+1) \rceil$, as each time the value of the ranking function decreases by k. For instance, if we replace φ_2 by $\varphi'_2 = \{i<n, i'=i*2\}$ and φ_3 by $\varphi'_3 = \{i<n, n'=n/2\}$ (and remove equation 4) then the method of [1] would infer that $\mathsf{I_r}$ is bound by $\lceil \log_k(\mathsf{nat}(n-i)+1) \rceil$.

Bounds on the Worst Cost of Equations. As it can be observed in the above example, in each application the corresponding equation might contribute a non-constant number of cost units. Therefore, it is not trivial to compute the worst-case (the maximum) value of all of them. In order to infer the maximum value of such expressions automatically, [1] proposes to first infer *invariants* (linear relations) between the equation's variables and the initial values. For example, the cost relation $C_m(i,n)$ admits as invariant for the recursive equations the formula $\mathcal{I}$ defined as $\mathcal{I}((i_0,n_0),(i,n)) \equiv i \geq i_0 \wedge n \leq n_0 \wedge i < n$, which captures that the values of i (resp. n) are greater (resp. smaller) or equal than the initial value and that i is smaller than n at all iterations. Once we have the invariant, we can *maximize* the expressions w.r.t. these values and take the maximal:

$$\mathit{worst}(\{\mathit{rec}_1, \ldots, \mathit{rec}_{nr}\}) = \max(\mathit{maximize}(\mathcal{I}, \{\mathit{rec}_1, \ldots, \mathit{rec}_{nr}\}))$$

The operator *maximize* receives an invariant $\mathcal{I}$ and a set of expressions to be maximized and computes the maximal value of each expression independently and returns the corresponding set of maximized expressions in terms of the initial values (see [1] for the technical details). For instance, in the original CR (without Eq. (4)), we compute $\mathit{worst}(\{\mathit{rec}_1, \mathit{rec}_2\}) = \max(\mathit{maximize}(\mathcal{I}, \{\mathsf{nat}(n-i), \mathsf{nat}(i)\}))$ which results in $\mathit{worst}(\{\mathit{rec}_1, \mathit{rec}_2\}) = \max(\{\mathsf{nat}(n_0 - i_0), \mathsf{nat}(n_0-1)\})$. The same procedure can be applied to the expressions in the base cases. However, it is unnecessary in our example, because the base case is a constant and therefore requires no maximization. Altogether, by applying Equation (*) to the standalone CR above we obtain the upper bounds shown in Ex. 1.

Inter-Procedural. In the above examples, all CRs are standalone and do not call any other equations. In the general case, a cost relation can contain k calls to external relations and n recursive calls: $\langle C(\bar{x}) = e + \sum_{i=1}^{k} D_i(\bar{y}_i) + \sum_{j=1}^{n} C(\bar{z}_j), \varphi \rangle$ with $k \geq 0$. After computing the upper bounds $D_i^+(\bar{y}_i)$ for the standalone CRs, we replace the computed upper bounds on the equations which call such relations, i.e., $\langle C(\bar{x}) = e + \sum_{i=1}^{k} D_i^+(\bar{y}_i) + \sum_{j=1}^{n} C(\bar{z}_j), \varphi \rangle$.

3 Asymptotic Notation for Cost Expressions

We now present extended versions of the standard definition of the asymptotic notations *big O* and *big Theta*, which handle functions with multiple input arguments, i.e., functions of the form $\mathbb{N}^n \mapsto \mathbb{R}^+$.

Definition 2 (big O, big Theta). *Given two functions $f, g : \mathbb{N}^n \mapsto \mathbb{R}^+$, we say that $f \in O(g)$ iff there is a real constant $c > 0$ and a natural constant $m \geq 1$ such that, for any $\bar{v} \in \mathbb{N}^n$ such that $v_i \geq m$, it holds that $f(\bar{v}) \leq c * g(\bar{v})$. Similarly, $f \in \Theta(g)$ iff there are real constants $c_1 > 0$ and $c_2 > 0$ and a natural constant $m \geq 1$ such that, for any $\bar{v} \in \mathbb{N}^n$ such that $v_i \geq m$, it holds that $c_1 * g(\bar{v}) \leq f(\bar{v}) \leq c_2 * g(\bar{v})$.*

The big O refers to asymptotic upper bounds and the big Θ to asymptotically tight estimates, when the asymptotic upper and lower bounds coincide. The asymptotic notations above assume that the value of the function increases with the values of the input such that the function, unless it has a constant asymptotic order, takes the value ∞ when the input is ∞. This assumption does not necessarily hold when CRs are obtained from realistic programs. For instance, consider the loop in Fig. 1. Clearly, the execution cost of the program increases by increasing the number of iterations of the loop, i.e., $n-i$, the ranking function. Therefore, in order to observe the limiting behavior of the program we should study the case when $\mathsf{nat}(n-i)$ goes to ∞, i.e., when, for example, n goes to ∞ and i stays constant, but not when both n and i go to ∞. In order to capture this asymptotic behaviour, we introduce the notion of nat-free cost expression, where we transform a cost expression into another one by replacing each nat-expression with a variable. This guarantees that we can make a consistent usage of the definition of asymptotic notation since, as intended, after some threshold m, larger values of the input variables result in larger values of the function.

Definition 3 (nat-free cost expressions). *Given a set of cost expression $E = \{e_1, \ldots, e_n\}$, the nat-free representation of E, is the set $\tilde{E} = \{\tilde{e}_1, \ldots, \tilde{e}_n\}$ which is obtained from E in four steps:*

1. *Each nat-expression $\mathsf{nat}(a_1x_1 + \cdots + a_nx_n + c) \in E$ which appears as an exponent is replaced by $\mathsf{nat}(a_1x_1 + \cdots + a_nx_n)$;*
2. *The rest of nat-expressions $\mathsf{nat}(a_1x_1 + \cdots + a_nx_n + c) \in E$ are replaced by $\mathsf{nat}(\frac{a_1}{b}x_1 + \cdots + \frac{a_n}{b}x_n)$, where b is the* greatest common divisor *(gcd) of $|a_1|, \ldots, |a_n|$, and $|\cdot|$ stands for the absolute value;*
3. *We introduce a fresh (upper-case) variable per syntactically different nat-expression.*
4. *We replace each nat-expression by its corresponding variable.*

Cases 1 and 2 above have to be handled separately because if $\mathsf{nat}(a_1x_1 + \cdots + a_nx_n + c)$ is an exponent, we can remove the c, but we cannot change the values of any a_i. E.g., $2^{\mathsf{nat}(2x+1)} \notin O(2^{\mathsf{nat}(x)})$. This is because $4^x \notin O(2^x)$. Hence, we cannot simplify $2^{\mathsf{nat}(2x)}$ to $2^{\mathsf{nat}(x)}$. In the case that $\mathsf{nat}(a_1x_1 + \cdots + a_nx_n + c)$ does not appear as an exponent, we can remove c and normalize all a_i by dividing them by the *gcd* of their absolute values. This allows reducing the number of variables which are needed for representing the nat-expressions. It is done by using just one variable for all nat expressions whose linear expressions are *parallel* and grow in the same direction. Note that removing the independent term plus dividing all constants by the gcd of their absolute values provides a canonical representation

for linear expressions. They satisfy this property iff their canonical representation is the same. This allows transforming both $\mathsf{nat}(2x{+}3)$ and $\mathsf{nat}(3x{+}5)$ to $\mathsf{nat}(x)$, and $\mathsf{nat}(2x{+}4y)$ and $\mathsf{nat}(3x{+}6y)$ to $\mathsf{nat}(x{+}2y)$.

Example 3. Given the following cost function:

$$5{+}7{*}\mathsf{nat}(3x+1){*}\max(\{100{*}\mathsf{nat}(x)^2{*}\mathsf{nat}(y)^4, 11{*}3^{\mathsf{nat}(y-1)}{*}\mathsf{nat}(x+5)^2\}){+}$$
$$2{*}\log(\mathsf{nat}(x+2)){*}2^{\mathsf{nat}(y-3)}{*}\log(\mathsf{nat}(y+4)){*}\mathsf{nat}(2x{-}2y)$$

Its nat-free representation is:

$$5{+}7*A{*}\max(\{100*A^2{*}B^4, 11*3^B{*}A^2\}){+}2{*}\log(A){*}2^B{*}\log(B){*}C$$

where A corresponds to $\mathsf{nat}(x)$, B to $\mathsf{nat}(y)$ and C to $\mathsf{nat}(x{-}y)$. □

Definition 4. *Given two cost expressions e_1, e_2 and its* nat*-free correspondence $\tilde{e}_1, \tilde{e}_2$, we say that $e_1{\in}O(e_2)$ (resp. $e_1{\in}\Theta(e_2)$) if $\tilde{e}_1{\in}O(\tilde{e}_2)$ (resp. $\tilde{e}_1{\in}\Theta(\tilde{e}_2)$).*

The above definition lifts Def. 2 to the case of cost expressions. Basically, it states that in order to decide the asymptotic relations between two cost expressions, we should check the asymptotic relation of their corresponding nat-free expressions. Note that by obtaining their nat-free expressions simultaneously we guarantee that the same variables are syntactically used for the same linear expressions.

In some cases, a cost expression might come with a set of constraints which specifies a class of input values for which the given cost expression is a valid bound. We refer to such set as *context constraint*. For example, the cost expression of Ex. 3 might have $\varphi{=}\{x{\geq}y, x{\geq}0, y{\geq}0\}$ as context constraint, which specifies that it is valid only for non-negative values which satisfy $x{\geq}y$. The context constraint can be provided by the user as an input to cost analysis, or collected from the program during the analysis.

The information in the context constraint φ associated to the cost expression can sometimes be used to check whether some nat-expressions are guaranteed to be asymptotically larger than others. For example, if the context constraint states that $x \geq y$, then when both $\mathsf{nat}(x)$ and $\mathsf{nat}(y)$ grow to the infinite we have that $\mathsf{nat}(x)$ asymptotically subsumes $\mathsf{nat}(y)$, this information might be useful in order to obtain more precise asymptotic bounds. In what follows, given two nat-expressions (represented by their corresponding nat-variables A and B), we say that $\varphi{\models}A \succeq B$ if A asymptotically subsumes B when both go to ∞.

4 Asymptotic Orders of Cost Expressions

As it is well-known, by using Θ we can partition the set of all functions defined over the same domain into *asymptotic orders*. Each of these orders has an infinite number of members. Therefore, to accomplish the motivations in Sect. 1 it is required to use one of the elements with simpler syntactic form. Finding a good representative of an asymptotic order becomes a complex problem when we deal with functions made up of non-linear expressions, exponentials, polynomials, and logarithms, possibly involving several variables and associated constraints. For example, given the cost expression of Ex. 3, we want to automatically infer the asymptotic order "$3^{\mathsf{nat}(y)} * \mathsf{nat}(x)^3$".

Apart from simple optimizations which remove constants and normalize expressions by removing parenthesis, it is essential to remove *redundancies*, i.e., subexpressions which are asymptotically subsumed by others, for the final expression to be as small as possible. This requires effectively comparing subexpressions of different lengths and possible containing multiple complexity orders. In this section, we present the basic definitions and a mechanism for transforming non-asymptotic cost expressions into non-redundant expressions while preserving the asymptotic order. Note that this mechanism can be used to transform the output of any cost analyzer into an non-redundant, asymptotically equivalent one. To the best of our knowledge, this is the first attempt to do this process in a fully automatic way. Given a cost expression e, the transformations are applied on its $\tilde{e}$ representation, and only afterwards we substitute back the nat-expressions, in order to obtain an asymptotic order of e, as defined in Def. 4.

4.1 Syntactic Simplifications on Cost Expressions

First, we perform some syntactic simplifications to enable the subsequent steps of the transformation. Given a nat-free cost expression $\tilde{e}$, we describe how to simplify it and obtain another nat-free cost expression $\tilde{e}'$ such that $\tilde{e} \in \Theta(\tilde{e}')$. In what follows, we assume that $\tilde{e}$ is not simply a constant or an arithmetic expression that evaluates to a constant, since otherwise we simply have $\tilde{e} \in O(1)$. The first step is to transform $\tilde{e}$ by removing constants and max expressions, as described in the following definition.

Definition 5. *Given a* nat*-free cost expression $\tilde{e}$, we denote by $\tau(\tilde{e})$ the cost expression that results from $\tilde{e}$ by: (1) removing all constants; and (2) replacing each subexpression $max(\{\tilde{e}_1, \ldots, \tilde{e}_m\})$ by $(\tilde{e}_1 + \ldots + \tilde{e}_m)$.*

Example 4. Applying the above transformation on the nat-free cost expression of Ex. 3 results in: $\tau(\tilde{e}) = A*(A^2*B^4 + 3^B*A^2) + \log(A)*2^B*\log(B)*C$. □

Lemma 1. $\tilde{e} \in \Theta(\tau(\tilde{e}))$

Once the τ transformation has been applied, we aim at a further simplification which safely removes sub-expressions which are asymptotically subsumed by other sub-expressions. In order to do so, we first transform a given cost expression into a *normal form* (i.e., a sum of products) as described in the following definition, where we use *basic* nat*-free cost expression* to refer to expressions of the form 2^{r*A}, A^r, or $\log(A)$, where r is a real number. Observe that, w.l.o.g., we assume that exponentials are always in base 2. This is because an expression n^A where $n > 2$ can be rewritten as $2^{\log(n)*A}$.

Definition 6 (normalized nat-free cost expression). *A* normalized nat*-free cost expression is of the form $\Sigma_{i=1}^{n} \Pi_{j=1}^{m_i} b_{ij}$ such that each b_{ij} is a basic* nat*-free cost expression.*

Since $b_1 * b_2$ and $b_2 * b_1$ are equal, it is convenient to view a product as the multiset of its elements (i.e., basic nat-free cost expressions). We use the letter M to

denote such multi-set. Also, since M_1+M_2 and M_2+M_1 are equal, it is convenient to view the sum as the multi-set of its elements, i.e., products (represented as multi-sets). Therefore, a normalized cost expression is a multi-set of multi-sets of basic cost expressions. In order to normalize a nat-free cost expression $\tau(\tilde{e})$ we will repeatedly apply the distributive property of multiplication over addition in order to get rid of all parenthesis in the expression.

Example 5. The normalized expression for $\tau(\tilde{e})$ of Ex. 4 is $A^3{*}B^4{+}2^{\log(3)*B}{*}$ $A^3{+}\ \log(A){*}2^B{*}\log(B)\ *\ C$ and its multi-set representation is $\{\{A^3, B^4\}$, $\{2^{\log(3)*B}, A^3\}$, $\{\log(A), 2^B, \log(B), C\}\}$ □

4.2 Asymptotic Subsumption

Given a normalized nat-free cost expression $\tilde{e} = \{M_1, \ldots, M_n\}$ and a context constraint φ, we want to remove from $\tilde{e}$ any product M_i which is *asymptotically subsumed* by another product M_j, i.e., if $M_j \in \Theta(M_j + M_i)$. Note that this is guaranteed by $M_i \in O(M_j)$. The remaining of this section defines a decision procedure for deciding if $M_i \in O(M_j)$. First, we define several *asymptotic subsumption templates* for which it is easy to verify that a single basic nat-free cost expression b subsumes a complete product. In the following definition, we use the auxiliary functions pow and deg of basic nat-free cost expressions which are defined as: $\mathsf{pow}(2^{r*A}) = r$, $\mathsf{pow}(A^r) = 0$, $\mathsf{pow}(\log(A)) = 0$, $\mathsf{deg}(A^r) = r$, $\mathsf{deg}(2^{r*A}) = \infty$, and $\mathsf{deg}(\log(A)) = 0$. In a first step, we focus on basic nat-free cost expression b with one variable and define when it asymptotically subsumes a set of basic nat-free cost expressions (i.e., a product). The product might involve several variables but they must be subsumed by the variable in b.

Lemma 2 (asymptotic subsumption). *Let b be a basic* nat*-free cost expression, $M = \{b_1, \cdots, b_m\}$ a product, φ a context constraint, $vars(b) = \{A\}$ and $vars(b_i) = \{A_i\}$. We say that M is asymptotically subsumed by b, i.e., $\varphi \models M \in O(b)$ if for all $1 \leq i \leq m$ it holds that $\varphi \models A \succeq A_i$ and one of the following holds:*

1. *if $b = 2^{r*A}$, then*
   (a) *$r > \Sigma_{i=1}^{m} \mathsf{pow}(b_i)$; or*
   (b) *$r \geq \Sigma_{i=1}^{m} \mathsf{pow}(b_i)$ and every b_i is of the form $2^{r_i*A_i}$;*
2. *if $b = A^r$, then*
   (a) *there is no b_i of the form $\log(A_i)$, then $r \geq \Sigma_{i=1}^{m} \mathsf{deg}(b_i)$; or*
   (b) *there is at least one b_i of the form $\log(A_i)$, and $r \geq 1 + \Sigma_{i=1}^{m} \mathsf{deg}(b_i)$*
3. *if $b = \log(A)$, then $m = 1$ and $b_1 = \log(A_1)$*

Let us intuitively explain the lemma. For exponentials, in point 1a, we capture cases such as $3^A = 2^{\log(3)*A}$ asymptotically subsumes $2^A * A^2 * \ldots * \log(A)$ where in "..." we might have any number of polynomial or logarithmic expressions. In 1b, we ensure that 3^A does not embed $3^A * A^2 * \log(A)$, i.e., if the power is the same, then we cannot have additional expressions. For polynomials, 2a captures that the largest degree is the upper bound. Note that an exponential would

introduce an ∞ degree. In 2b, we express that there can be many logarithms and still the maximal polynomial is the upper bound, e.g., A^2 subsumes $A * \log(A) * \log(A) * \ldots * \log(A)$. In 3, a logarithm only subsumes another logarithm.

Example 6. Let $b = A^3$, $M = \{\log(A), \log(B), C\}$, where A, B and C corresponds to $\mathsf{nat}(x)$, $\mathsf{nat}(y)$ and $\mathsf{nat}(x-y)$ respectively. Let us assume that the context constraint is $\varphi = \{x \geq y, x \geq 0, y \geq 0\}$. M is asymptotically subsumed by b since $\varphi \models (A \succeq B) \wedge (A \succeq C)$, and condition 2b in Lemma 2 holds. □

The basic idea now is that, when we want to check the subsumption relation on two expression M_1 and M_2 we look for a partition of M_2 such that we can prove the subsumption relation of each element in the partition by a different basic nat-free cost expression in M_1. Note that M_1 can contain additional basic nat-free cost expressions which are not needed for subsuming M_2.

Lemma 3. *Let M_1 and M_2 be two products, and φ a context constraint. If there exists a partition of M_2 into k sets $P_1, \ldots, P_k$, and k distinct basic* nat*-free cost expressions $b_1, \ldots, b_k \in M_1$ such that $P_i \in O(b_i)$, then $M_2 \in O(M_1)$.*

Example 7. Let $M_1 = \{2^{\log(3)*B}, A^3\}$ and $M_2 = \{\log(A), 2^B, \log(B), C\}$, with the context constraint φ as defined in Ex. 6. If we take $b_1 = 2^{\log(3)*A}$, $b_2 = A^3$, and partition M_2 into $P_1 = \{2^B\}$, $P_2 = \{\log(A), \log(B), C\}$ then we have that $P_1 \in O(b_1)$ and $P_2 \in O(b_2)$. Therefore, by Lemma 3, $M_2 \in O(M_1)$. Also, for $M_2' = \{A^3, B^4\}$ we can partition it into $P_1' = \{B^4\}$ and $P_2' = \{A^3\}$ such that $P_1' \in O(b_1)$ and $P_2' \in O(b_2)$ and therefore we also have that $M_2' \in O(M_1)$. □

Definition 7 (asymp). *Given a cost expression e, the overall transformation* asymp *takes e and returns the cost expression that results from removing all subsumed products from the normalized expression of $\tau(\tilde{e})$, and then replace each* nat*-variable by the corresponding* nat*-expression.*

Example 8. Consider the normalized cost expression of Ex. 5. The first and third products can be removed, since they are subsumed by the second one, as explained in Ex. 7. Then $\mathtt{asymp}(e)$ would be $2^{\log(3)*\mathsf{nat}(y)} * \mathsf{nat}(x)^3 = 3^{\mathsf{nat}(y)} * \mathsf{nat}(x)^3$, and it holds that $e \in \Theta(\mathtt{asymp}(e))$. □

In the following theorem, we ensure that after eliminating the asymptotically subsumed products, we preserve the asymptotic order.

Theorem 1 (soundness). *Given a cost expression e and a context constraint φ, then $\varphi \models e \in \Theta(\mathtt{asymp}(e))$.*

4.3 Implementation in COSTA

We have implemented our transformation and it can be used as a back-end of existing non-asymptotic cost analyzers for average, lower and upper bounds (e.g., [9,2,12,5,7]), and regardless of whether it is based on the approach to cost analysis of [15] or any other. We plan to distribute it as free software soon.

Currently, it can be tried out through a web interface available from the COSTA web site: `http://costa.ls.fi.upm.es`. COSTA is an abstract interpretation-based COSt and Termination Analyzer for Java bytecode which receives as input a bytecode program and (a choice of) a resource of interest, and tries to obtain an upper bound of the resource consumption of the program.

In our first experiment, we use our implementation to obtain asymptotic forms of the upper bounds on the memory consumption obtained by [4] for the JOlden suite [10]. This benchmark suite was first used by [6] in the context of memory usage verification and is becoming a standard to evaluate memory usage analysis [5,4]. None of the previous approaches computes asymptotic bounds. We are able to obtain accurate asymptotic forms for all benchmarks in the suite and the transformation time is negligible (less than 0.1 milliseconds in all cases). As a simple example, for the benchmark `em3d`, the non-asymptotic upper bound is $8{*}\mathsf{nat}(d{-}1){*}\mathsf{nat}(b){+}8{*}\mathsf{nat}(d){+}8{*}\mathsf{nat}(b)\ {+}56{*}\mathsf{nat}(d{-}1){+}16{*}\mathsf{nat}(c)\ {+}73$ and we transform it to $\mathsf{nat}(d){*}\mathsf{nat}(b){+}\mathsf{nat}(c)$. The remaining examples can be tried online in the above `url`.

5 Generation of Asymptotic Upper Bounds

In this section we study how to perform a tighter integration of the asymptotic transformation presented Sec. 4 within resource usage analyses which follow the classical approach to static cost analysis by Wegbreit [15]. To do this, we reformulate the process of inferring upper bounds sketched in Sect. 2.2 to work directly with asymptotic functions at all possible (intermediate) stages. The motivation for doing so is to reduce the huge amount of memory required for constructing non-asymptotic bounds and, in the limit, to be able to infer asymptotic bounds in cases where their non-asymptotic forms cannot be computed.

Asymptotic CRS. The first step in this process is to transform cost relations into asymptotic form before proceeding to infer upper bounds for them. As before, we start by considering standalone cost relations. Given an equation of the form $\langle C(\bar{x}){=}e{+}\sum_{i=1}^{k} C(\bar{y}_i), \varphi\rangle$ with $k \geq 0$, its associated *asymptotic* equation is $\langle C_A(\bar{x}){=}\mathtt{asymp}(e){+}\sum_{i=1}^{k} C_A(\bar{y}_i), \varphi\rangle$. Given a cost relation C, its asymptotic cost relation C_A is obtained by applying the above transformation to all its equations. Applying the transformation at this level is interesting in order to simplify both the process of computing the worst case cost of the recursive equations and the base cases when computing Eq. $(*)$ as defined in Sect. 2.2.

Example 9. Consider the following CR:

$$\langle C(a,b) = \underline{\mathsf{nat}(a+1)^2}\ ,\ \{a{\geq}0, b{\geq}0\}\rangle$$
$$\langle C(a,b) = \underline{\mathsf{nat}(a{-}b){+}\log(\mathsf{nat}(a{-}b))}{+}C(a',b')\ ,\ \{a{\geq}0, b{\geq}0, a'{=}a{-}2, b'{=}b{+}1\}\rangle$$
$$\langle C(a,b) = \underline{2^{\mathsf{nat}(a+b)}{+}\mathsf{nat}(a){*}\log(\mathsf{nat}(a))}{+}C(a',b')\ ,\ \{a{\geq}0, b{\geq}0, a'{=}a{+}1, b'{=}b{-}1\}\rangle$$

By replacing the underlined expressions by their corresponding `asymp` expressions as explained in Theorem 1, we obtain the asymptotic relation:

$\langle C_A(a,b) = \mathsf{nat}(a)^2 \ , \ \{a{\geq}0, b{\geq}0\}\rangle$
$\langle C_A(a,b) = \mathsf{nat}(a{-}b){+}C_A(a',b') \ , \ \{a{\geq}0, b{\geq}0, a'{=}a{-}2, b'{=}b{+}1\}\rangle$
$\langle C_A(a,b) = 2^{\mathsf{nat}(a+b)}{+}C_A(a',b') \ , \ \{a{\geq}0, b{\geq}0, a'{=}a{+}1, b'{=}b{-}1\}\rangle$

In addition to reducing their sizes, the process of maximizing the nat expressions is more efficient since there are fewer nat expressions in the asymptotic CR. □

An important point to note is that, while we can remove all constants from e, it is essential that we keep the constants in the size relations φ to ensure soundness. This is because they are used to infer the ranking functions and to compute the invariants, and removing such constants might introduce imprecision and more important soundness problems as we explain in the following examples.

Example 10. The above relation admits a ranking function $f(a,b){=}\mathsf{nat}(2a + 3b{+}1)$ which is used to bound the number of applications of the recursive equations. Clearly, if we remove the constants in the size relations, e.g., transform $a'{=}a{-}2$ into $a'{=}a$, the resulting relation is non-terminating and we cannot find a ranking function. Besides, removing constants from constraints which are not necessarily related to the ranking function also might result in incorrect invariants. For example, changing $n'{=}n{+}1$ to $n'{=}n$ in the following equation:

$$\langle C(m,n) = \mathsf{nat}(n) + C(m',n') \ , \ \{m{>}0, m'{<}m, n'{=}n{+}1\}\rangle$$

would result in an invariant which states that the value of n is always equal to the initial value n_0, which in turn leads to the upper-bound $\mathsf{nat}(m_0){*}\mathsf{nat}(n_0)$ which is clearly incorrect. A possible correct upper-bound is $\mathsf{nat}(m_0){*}\mathsf{nat}(n_0 + m_0)$ which captures that the value of $\mathsf{nat}(n)$ increases up to $\mathsf{nat}(n_0{+}m_0)$. □

Asymptotic Upper Bounds. Once the standalone CR is put into asymptotic form, we proceed to infer an upper bound for it as in the case of non-asymptotic CRs and then we apply the transformation to the result. Let $C_A(\bar{x})$ be an asymptotic cost relation. Let $C_A^+(\bar{x})$ be its upper bound computed as defined in Eq. $(*)$. Its asymptotic upper bound is $C^+_{asymp}(\bar{x}) = \mathtt{asymp}(C_A^+(\bar{x}))$. Observe that we are computing $C_A^+(\bar{x})$ in a non-asymptotic fashion, i.e., we do not apply asymp to each $\mathsf{l_b}$, $\mathsf{l_r}$, *worst* in $(*)$, but only to the result of combining all elements. We could apply asymp to the individual elements and then to the result of their combination again. In practice, it almost makes no difference as this operation is really inexpensive.

Example 11. Consider the second CR of Ex. 9. The analyzer infers the invariant $\mathcal{I} = \{0{\leq}a{\leq}a_0, 0{\leq}b{\leq}b_0, a{\geq}0, b{\geq}0\}$, from which we maximize $\mathsf{nat}(a)^2$ to $\mathsf{nat}(a_0)^2$, $\mathsf{nat}(a{-}b)$ to $\mathsf{nat}(a_0)$ (since the maximal value occurs when b becomes 0), and $2^{\mathsf{nat}(a+b)}$ to $2^{\mathsf{nat}(a_0+b_0)}$. The number of applications of the recursive equations is $\mathsf{nat}(2a_0{+}3b_0{+}1)$ (see Ex. 10). By applying Eq. $(*)$, we obtain the upper bound: $C_A^+(a,b) = \mathsf{nat}(2a{+}3b{+}1) * \max(\{\mathsf{nat}(a), 2^{\mathsf{nat}(a+b)}\}) + \mathsf{nat}(a)^2$. Applying asymp to the above upper bound results in: $C^+_{asymp}(a,b) = 2^{\mathsf{nat}(a+b)} * \mathsf{nat}(2a + 3b)$. □

Inter-procedural. The practical impact of integrating the asymptotic transformation within the solving method comes when we consider relations with

calls to external relations and compose their asymptotic results. This is because, when the number of calls and equations grow, the fact that we manipulate more compact asymptotic expressions is fundamental to enable the scalability of the system. Consider a cost relation with k calls to external relations and n recursive calls: $\langle C(\bar{x})=e+\sum_{i=1}^{k} D_i(\bar{y}_i)+\sum_{j=1}^{n} C(\bar{z}_j), \varphi\rangle$ with $k \geq 0$. Let $D^{+}_{i_{asymp}}(\bar{y}_i)$ be the asymptotic upper bound for $D_i(\bar{y}_i)$. $C^{+}_{asymp}(\bar{x})$ is the asymptotic upper bound of the standalone relation $\langle C(\bar{x})=e+\sum_{i=1}^{k} D^{+}_{i_{asymp}}(\bar{y}_i)+\sum_{j=1}^{n} C(\bar{z}_j), \varphi\rangle$.

Theorem 2 (soundness). $C^{+}(\bar{x}) \in O(C^{+}_{asymp}(\bar{x}))$.

Note that the soundness theorem, unlike Th. 1, guarantees only that the asymptotic expression is O and not Θ. Let us show an example.

Example 12. Consider ub=$\mathsf{nat}(a-b+1)*2^{\mathsf{nat}(c)}+5$ and $\mathtt{asymp}(ub)$=$\mathsf{nat}(a-b)*2^{\mathsf{nat}(c)}$. Plugging ub in a context where b=a+1 results in 5 (since then $\mathsf{nat}(a-b+1)$ =0). Plugging $\mathtt{asymp}(ub)$ in the same context results in $2^{\mathsf{nat}(c)}$ which is clearly less precise. □

Intuitively, the source of the loss of precision is that, when we compute the asymptotic upper bound, we are looking at the cost in the limiting behavior only and we might miss a particular point in which such cost becomes zero. In our experience, this does not happen often and it could be easily checked before plugging in the asymptotic result, replacing the upper bound by zero.

5.1 Experimental Results on Scalability

In this section, we aim at studying how the size of cost expressions (non-asymptotic vs. asymptotic) increases when larger CRs are used, i.e., the scalability of our approach. To do so, we have used the benchmarks of [1] shown in Table 1. These benchmarks are interesting because they cover the different complexity order classes, as it can be seen, the benchmarks range from constant to exponential complexity, including polynomial and divide and conquer. The source code of such programs is also available at the COSTA web site.

As in [1], in order to assess the scalability of the approach, we have connected together the CRs for the different benchmarks by introducing a call from each CR to the one appearing immediately above it in the table. Such call is always introduced in a recursive equation. Column **#Eq** shows the number of equations in the corresponding benchmarks. Reading this column top-down, we can see that when we analyze BST we have 31 equations. Then, for Fibonacci, the number of equations is 39, i.e., its 8 equations plus the 31 which have been previously accumulated. Progressively, each benchmark adds its own number of equations to the one above. Thus, in the last row we have a CR with all the equations connected, i.e., we compute an upper bound of a CR with at least 20 nested loops and 385 equations.

Columns $\mathbf{T}_{ub}$ and $\mathbf{T}_{aub}$ show, respectively, the times of composing the non-asymptotic and asymptotic bounds, after discarding the time common part for both, i.e., computing the ranking functions and the invariants. It can be observed

Table 1. Scalability of asymptotic cost expressions

Bench.	$\mathbf{T}_{ub}$	$\mathbf{T}_{aub}$	$\mathbf{Size}_{ub}$	$\mathbf{Size}_{aub}$	**#Eq**	$\frac{\mathbf{Size}_{ub}}{\mathbf{\#Eq}}$	$\frac{\mathbf{Size}_{aub}}{\mathbf{\#Eq}}$	$\frac{\mathbf{Size}_{ub}}{\mathbf{Size}_{aub}}$
BST	0	0	23	4	31	0.74	0.13	5.75
Fibonacci	0	0	47	9	39	1.21	0.23	5.22
Hanoi	0	0	67	14	48	1.39	0.29	4.78
MatMult	0	0	152	38	67	2.27	0.56	4.00
Delete	0	4	320	65	100	3.20	0.65	4.92
FactSum	4	4	717	95	117	6.12	0.81	7.54
SelectOrd	0	4	1447	155	136	10.63	1.14	9.33
ListInter	4	16	3804	257	173	21.98	1.48	14.80
EvenDigits	4	20	7631	400	191	39.95	2.09	19.07
Cons	12	32	15268	585	214	71.34	2.73	26.09
Power	24	40	24265	588	223	108.81	2.63	41.26
MergeList	96	60	48536	828	245	198.10	3.37	58.61
ListRev	140	76	48545	829	254	191.12	3.26	58.55
Incr	×	112	×	1126	282	×	3.99	×
Concat	×	164	×	1538	296	×	5.19	×
ArrayRev	×	232	×	2127	305	×	6.97	×
Factorial	×	284	×	2130	314	×	6.78	×
DivByTwo	×	328	×	2135	323	×	6.60	×
Polynomial	×	436	×	2971	346	×	8.58	×
MergeSort	×	440	×	3234	385	×	8.40	×

that the times are negligible from BST to EvenDigits, which are the simplest benchmarks and also have few equations. The interesting point is that when cost expressions start to be considerably large, $\mathbf{T}_{ub}$ grows significantly, while $\mathbf{T}_{aub}$ remains small. This is explained by the sizes of the expressions they handle, as we describe below. For the columns that contain "×", COSTA has not been able to compute a non-asymptotic upper bound because the underlying Prolog process has run out of memory.

Columns $\mathbf{Size}_{ub}$ and $\mathbf{Size}_{aub}$ show, respectively, the sizes of the computed non-asymptotic and asymptotic upper bounds. This is done by regarding the upper bound expression as a tree and counting its number of nodes, i.e., each operator and each operand is counted as one. As for the time, the sizes are quite small for the simplest benchmarks, and they start to increase from SelectOrd. Note that for these examples, the size of the non-asymptotic upper bounds is significantly larger than the asymptotic. Columns $\frac{\mathbf{Size}_{ub}}{\mathbf{\#Eq}}$ and $\frac{\mathbf{Size}_{aub}}{\mathbf{\#Eq}}$ show, resp., the size of the non-asymptotic and asymptotic bounds per equation. The important point is that while this ratio seems to grow exponentially for non-asymptotic upper bounds, $\frac{\mathbf{Size}_{aub}}{\mathbf{\#Eq}}$ grows much more slowly. We believe that this demonstrates that our approach is scalable, even if the implementation is still preliminary.

6 Conclusions and Future Work

We have presented a general asymptotic resource usage analysis which can be combined with existing non-asymptotic analyzers by simply adding our transformation as a back-end or, interestingly, integrated into the mechanism for obtaining upper bounds of recurrence relations. This task has been traditionally done manually in the context of complexity analysis. When it comes to apply it to an automatic analyzer for a real-life language, there is a need to develop the techniques to infer asymptotic bounds in a precise and effective way. To the best of our knowledge, our work is the first one which presents a generic and fully automatic approach. In future work, we plan to adapt our general framework to infer asymptotic lower-bounds on the cost and also to integrate our work into a proof-carrying code infrastructure.

Acknowledgments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-231620 *HATS* project, by the MEC under the TIN-2008-05624 *DOVES* and HI2008-0153 (Acción Integrada) projects, by the UCM-BSCH-GR58/08-910502 (GPD-UCM) , and the CAM under the S-0505/ TIC/0407 *PROMESAS* project.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 221–237. Springer, Heidelberg (2008)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 113–132. Springer, Heidelberg (2008)
4. Albert, E., Genaim, S., Gómez-Zamalloa, M.: Live Heap Space Analysis for Languages with Garbage Collection. In: ISMM. ACM Press, New York (2009)
5. Braberman, V., Fernández, F., Garbervetsky, D., Yovine, S.: Parametric Prediction of Heap Memory Requirements. In: ISMM. ACM Press, New York (2008)
6. Chin, W.-N., Nguyen, H.H., Qin, S., Rinard, M.C.: Memory Usage Verification for OO Programs. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 70–86. Springer, Heidelberg (2005)
7. Chin, W.-N., Nguyen, H.H., Popeea, C., Qin, S.: Analysing Memory Resource Bounds for Low-Level Programs. In: ISMM. ACM Press, New York (2008)
8. Debray, S.K., Lin, N.W.: Cost analysis of logic programs. TOPLAS 15(5) (1993)
9. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: Speed: precise and efficient static estimation of program computational complexity. In: POPL, pp. 127–139. ACM, New York (2009)
10. JOlden Suite Collection, `http://www-ali.cs.umass.edu/DaCapo/benchmarks.html`

11. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall, New York (1993)
12. Navas, J., Méndez-Lojo, M., Hermenegildo, M.: User-Definable Resource Usage Bounds Analysis for Java Bytecode. In: BYTECODE. Elsevier, Amsterdam (2009)
13. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
14. Sands, D.: Complexity Analysis for a Lazy Higher-Order Language. In: Jones, N.D. (ed.) ESOP 1990. LNCS, vol. 432, pp. 361–376. Springer, Heidelberg (1990)
15. Wegbreit, B.: Mechanical Program Analysis. Comm. of the ACM 18(9) (1975)

A Properties of O and Θ

In this section we first review some properties of the asymptotics orders O and Θ. Abusing of notation, in the following we use $e_1, e_2, ...$ to refer to nat-free cost expressions. Now, given a nat-free cost expression e such that $vars(e) \subseteq \{A_1, \ldots, A_n\}$ and an assignment f from $\{A_i \ldots, A_n\}$ to $\mathbb{R}^+$ such that $f(A_i) = v_i$, we denote by $f(e)$ to the result of evaluating the expression e by replacing in e each occurrence of A_i by v_i.

Definition 8. *Let e_1, e_2 be two nat-free cost expressions such that $vars(e_1) \cup vars(e_2) = \{A_1, \ldots, A_n\}$. Then, we say that*

- *e_1 and e_2 are equivalent, written $e_1 = e_2$, if and only if for any assignment f from $\{A_1, \ldots, A_n\}$ to $\mathbb{R}^+$ it holds that $f(e_1) = f(e_2)$.*
- *e_1 is less or equal than e_2, written $e_1 \leq e_2$, if and only if for any assignment f from $\{A_1, \ldots, A_n\}$ to $\mathbb{R}^+$ it holds that $f(e_1) \leq f(e_2)$.*

Abstract properties

Property 1. General properties of O and Θ. Let e_1, e_2, e_3 be three nat-free cost expressions:

1. $e_1 \leq e_2 \Rightarrow e_1 \in O(e_2)$.
2. $e_1 = e_2 \Rightarrow e_1 \in \Theta(e_2)$.
3. $e_1 \in O(e_1)$ and $e_1 \in \Theta(e_1)$.
4. $e_1 \in \Theta(e_2) \Leftrightarrow e_2 \in \Theta(e_1)$.
5. $e_1 \in \Theta(e_2) \wedge e_2 \in \Theta(e_3) \Rightarrow e_1 \in \Theta(e_3)$.
6. $e_1 \in \Theta(e_2) \Leftrightarrow e_1 \in O(e_2) \wedge e_2 \in O(e_1)$.
7. $e_1 \in O(e_2) \wedge e_2 \in O(e_3) \Rightarrow e_1 \in O(e_3)$.

Property 2. Properties related to constants:

- $\forall r, s \in \mathbb{R}^+$ it holds that $r \in \Theta(s)$.
- $\forall k \in \mathbb{R}^+$ it holds that $e * k \in \Theta(e)$.

Sums, max, products

Property 3. Let e_1, e_2, e_3, e_4 be nat-free cost expressions. Then

1. If $e_1 \in O(e_3)$ and $e_2 \in O(e_4)$ then $e_1 + e_2 \in O(e_3 + e_4)$.
2. If $e_1 \in \Theta(e_3)$ and $e_2 \in \Theta(e_4)$ then $e_1 + e_2 \in \Theta(e_3 + e_4)$.
3. If $e_1 \in O(e_2)$ then $e_1 + e_2 \in \Theta(e_2)$.
4. If $e > 0$ then $\forall r \in \mathbb{R}^+$ it holds that $e + r \in \Theta(e)$.

Property 4. Let $\{e_1, ..., e_n\}$ be a set of nat-free cost expressions. Then it holds that $\max\{e_1, \ldots, e_n\} \in \Theta\left(\sum_{i=1}^{n} e_i\right)$. Note that this property holds trivially since $\max\{e_1, \ldots, e_n\} \leq \sum_{i=1}^{n} e_i \leq n * max\{e_1, \ldots, e_n\}$.

Property 5. Let e_1, e_2, e_3, e_4, e be nat-free cost expressions. Then it holds:

1. $e_1 \in O(e_3) \wedge e_2 \in O(e_4) \Rightarrow e_1 * e_2 \in O(e_3 * e_4)$.
2. $e_1 \in \Theta(e_3) \wedge e_2 \in \Theta(e_4) \Rightarrow e_1 * e_2 \in \Theta(e_3 * e_4)$.

Logarithms and powers

Property 6. Let e_1, e_2 be two nat-free cost expressions. Then it holds that:

1. If $e_1 \in O(e_2)$ then $\forall r \in \mathbb{R}^+$ it holds that $e_1^r \in O(e_2^r)$.
2. If $e_1 \in \Theta(e_2)$ then $\forall r \in \mathbb{R}^+$ it holds that $e_1^r \in \Theta(e_2^r)$.

Property 7. For every $r, s \in \mathbb{R}^+$, $r, s > 0$, if $r \leq s$ then $e^r \in O(e^s)$.

Property 8. If $e_1 \in O(e_2)$ then $\log(e_1) \in O(\log(e_2))$.

Property 9. For every $r \in \mathbb{R}^+$, $r > 0$, it holds that $\log(e) \in O(e^r)$. Therefore, if $r, s \in \mathbb{R}^+$, $r, s > 0$, then it holds that $\log(e)^s \in O(e^r)$.

Property 10. Let $e_1, ..., e_n, e, e'$ be nat-free cost expressions and let φ be a linear constraint such that $\varphi \models e_i \geq 1$, $\varphi \models e_i \leq e$, $1 \leq i \leq n$. Then $\forall r, s_1, ..., s_n, t \in \mathbb{R}^+$ such that $r > s_1 + ... + s_n$ it holds that

$$\varphi \models \prod_{i=1}^{n} e_i^{s_i} * log^t(e') \in O\left(e^r\right)$$

Proof. If we descompose $e^r = e^{r'} * e^{s'}$ where $s' = \sum_{i=1}^{n} s_i$ and $r' = r - s'$, we get that $r' > 0$ and therefore we can apply Property 3 because

- $\varphi \models \prod_{i=1}^{n} e_i^{s_i} \leq e^{s'}$.
- $\log(e')^t \in O(e^{r'})$ by application of Property 9.

Property 11. For any $r, s \in \mathbb{R}^+$, $s > 0$, it holds that $e^r \in O\left(2^{s*e}\right)$.

Property 12. Let $e_1, ..., e_n, e, e'$ be nat-free cost expressions and let φ a linear constraint such that $\varphi \models 0 \leq e_i \leq e$. Then $\forall r, s_1, ..., s_n, t \in \mathbb{R}^+$ such that $r > s_1 + ... + s_n$, it holds that

$$\varphi \models 2^{\sum_{i=1}^{n} s_i * e_i} * e'^t \in O\left(2^{r*e}\right)$$

Proof. First, we descompose $2^{r*e} = 2^{r'*e} * 2^{s'*e}$ where $s' = \sum_{i=1}^{n} s_i$ and $r' = r - s' > 0$. Using Property 3 it's enough to see that:

1. $\sum_{i=1}^{n} s_i * e_i \leq s' * e$.
2. $(e')^t \in O(2^{r'*e})$ as a consequence of Properties 9 and 11.

B Proofs

B.1 Proof of Lemma 1

Proof. We must proof that given any nat-free cost expression e, it holds that $e \in \Theta(\tau(e))$. To this end it is enough to take into accout that:

- For all subexpression of the form $\max\{e_1, ..., e_n\}$ in e, it holds (thanks to Property 4) that $\max\{e_1, ..., e_n\} \in \Theta(e_1 + ... + e_n)$.
- For the remaining operations, the result follows from Properties 3 (sum), 1 (product), 6 (power) and 8 (logarithm).

B.2 Proof of Lemma 2

Proof. Being $M = b_1 * ... * b_n$, we must proof that $M \in O(b)$. Let us define the numbers $p = \sum_{i=1}^{n} \mathsf{pow}(b_i)$ and $g = \sum_{i=1}^{m} \mathsf{deg}(b_i)$. Then, we distinguish the following three cases:

1. Exponential: Suppose that $b = 2^{r*A}$, for some $r \in \mathbb{R}^+$. Then:
   (a) If $r > p$ then Property 12 entails the result.
   (b) If $r \geq p$ and all b_i are of the form $2^{r_i * A_i}$, where $r_i = \mathsf{pow}(b_i)$, then we can use that
   $$\varphi \models r * A \geq \sum_{i=1}^{n} r_i * A_i$$
   in order to infer $b = 2^{r*A} \geq 2^{p*A}$.
2. Polynomial: Assume $b = A^r$ for some $r \in \mathbb{R}^+$ and $p = 0$. Then:
   (a) If $r > g$, the result follows from Property 10.
   (b) If $r \geq g$ and there are no b_i being logarithmic, we only have to consider that
   $$b^r \geq \prod_{i=1}^{n} b_i^{g_i}$$
3. Logarithmic: If $e = \log(A)$ and $M = \log(A_i)$, then by Property 8 it holds that $\log(A) \leq \log(A_i)$.

B.3 Proof of Lemma 3

Proof. Let $M_1 = \{b_1, ..., b_m\}$, $M_2 = \{b'_1, ..., b'_n\}$ be two products. Let us consider the following partition in $M_2 = \bigcup_{i=1}^{k} S_i$, where $k \leq n$, $k \leq m$. Let $f : \{1, \ldots, k\} \to \{1, \ldots, m\}$ be the index injective function. It holds that if $\forall i \in \{1, \ldots, k\} : M_i \in O\left(b_{f(i)}\right)$ then $M_2 \in O(M_1)$. Note that if M_2 admits a factorization $M_2 = S_1 * ... * S_k$ and each S_i has its own unique b_j such that $S_i \in O(b_j)$, then from Property 1, it holds $S_1 * ... * S_k \in O(b_{f(1)} * ... * b_{f(k)})$. Hence $M_2 \in O(M_1)$.

B.4 Proof of Theorem 1

Proof. Let us suppose that e is equal to $m_1 + \ldots + m_{n+k}$ and that the last k $(m_{n+1}, \ldots, m_{n+k})$ elements are subsumed by the first n $(m_1, \ldots, m_n)$. If we define:

$$e' = \mathtt{asymp}(e) = \sum_{i=1}^{n} m_i \qquad\qquad e = e' + \sum_{i=1}^{k} m_{n+i}$$

If for all m_{n+i} it holds that $m_i \in O(e')$, then we can apply Property 3.3 and get that $e' + \sum_{i=1}^{k} m_{n+i} \in \Theta(e')$.

B.5 Proof of Theorem 2

Proof. Let $C(\overline{x})$ be a cost relation defined as

$$\langle C(\overline{x}) = e + \sum_{i=1}^{k} D_i(\overline{y}_i) + \sum_{i=1}^{n} C(\overline{z}), \varphi \rangle$$

such that for every external cost relation $D_i(\overline{w})$ there is a known cost expression $D^+_{i_{asymp}}(\bar{w})$ such that $D_i(\bar{w}) \in O(D^+_{i_{asymp}}(\bar{w}))$. We can define $C_{asymp}(\overline{x})$ as

$$\langle C_{asymp}(\overline{x}) = e + \sum_{i=1}^{k} D^+_{i_{asymp}}(\bar{y}_i) + \sum_{i=1}^{n} C(\overline{z}), \varphi \rangle$$

In order to prove $C^+(\overline{x}) \in O(C^+_{asymp}(\overline{x}))$ it is enough to prove that

$$e + \sum_{i=1}^{k} D_i(\overline{y}_i) \in O\left(e + \sum_{i=1}^{k} D^+_{i_{asymp}}(\bar{y}_i)\right)$$

which can be done by induction on the number k of external calls.

- Base case $k = 0$. In this case, it becomes $e \in O(e)$ which is a trivial property of O.
- Inductive case: if we suppose that

  $$e + \sum_{i=1}^{k-1} D_i(\overline{y}_i) \in O\left(e + \sum_{i=1}^{k-1} D^+_{i_{asymp}}(\overline{y}_i)\right)$$

  and we have that

  $$\sum_{i=1}^{k} D_i(\overline{y}_i) = \sum_{i=1}^{k-1} D_i(\overline{y}_i) + D_k(\overline{y}_k)$$
  $$\sum_{i=1}^{k} D_{i_{asymp}}(\overline{y}_i) = \sum_{i=1}^{k-1} D_{i_{asymp}}(\overline{y}_i) + D_{k_{asymp}}(\overline{y}_k)$$

  so we can apply Property 3 and infer that

  $$e + \sum_{i=1}^{k-1} D_i(\overline{y}_i) \in O\left(e + \sum_{i=1}^{k} D^+_{i_{asymp}}(\overline{y}_i)\right)$$

Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 279 (1) (2011) 3–17

www.elsevier.com/locate/entcs

Handling Non-linear Operations in the Value Analysis of COSTA

Diego Alonso [a] Puri Arenas[a] Samir Genaim[a]

[a] *DSIC, Complutense University of Madrid (UCM), Spain*

Abstract

Inferring precise relations between (the values of) program variables at different program points is essential for termination and resource usage analysis. In both cases, this information is used to synthesize ranking functions that imply the program's termination and bound the number of iterations of its loops. For efficiency, it is common to base *value analysis* on non-disjunctive abstract domains such as Polyhedra, Octagon, etc. While these domains are efficient and able to infer complex relations for a wide class of programs, they are often not sufficient for modeling the effect of non-linear and bit arithmetic operations. Modeling such operations precisely can be done by using more sophisticated abstract domains, at the price of performance overhead. In this paper we report on the value analysis of COSTA that is based on the idea of encoding the disjunctive nature of non-linear operations into the (abstract) program itself, instead of using more sophisticated abstract domains. Our experiments demonstrate that COSTA is able to prove termination and infer bounds on resource consumption for programs that could not be handled before.

Keywords: Resource usage analysis, value analysis, non-linear operations, bit arithmetic operations

1 Introduction

Termination and resource usage analysis of imperative languages have received a considerable attention [3,22,20,8,19,13,14]. Most of these analyses rely on a value (or size) analysis component, which infers relations between the values of the program variables (or the sizes of the corresponding data structures) at different program points. This information is then used to bound the number of iterations of the program's loops. Thus, the precision of value analysis directly affects the class of (terminating) programs for which the corresponding tool is able prove termination or infer lower and upper bounds on their resource consumption. Moreover, in the case of resource consumption, it also affects the quality of the inferred bounds (i.e., how tight there are).

Typically, for efficiency, the underlying abstract domains used in value analysis are based on conjunctions of linear constraints, e.g., Polyhedra [10], Octagons [18], etc. While in practice these abstract domains are precise enough for bounding the loops of many programs, they are often not sufficient when the considered program

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2011.11.002

involves non-linear arithmetic operations (multiplication, division, bit arithmetics, etc). This is because the semantics of such operations cannot be modeled precisely with only conjunctions of linear constraints. In order to overcome this limitation, one can use abstract domains that support non-linear constraints, however, these domain typically impose a significant performance overhead. Another alternative is to use disjunctive abstract domains, i.e., disjunctions of (conjunctions of) linear constraints. This allows splitting the behavior of the corresponding non-linear operation into several mutually exclusive cases, such that each one can be precisely described using only conjunctions of linear constraints. This alternative also imposes performance overhead, since the operations of such disjunctive abstract domains are usually more expensive.

In this paper, we develop a value analysis that handles non-linear arithmetic operations using disjunctions of (conjunctions of) linear constraints. However, similarly to [21], instead of directly using disjunctive abstract domains, we encode the disjunctive nature of the non-linear operations directly in the (abstract) program. This allows using non-disjunctive domains like Polyhedra, Octagons, etc., and still benefit from the disjunctive information in order to infer more precise relations for programs with non-linear arithmetic operations. We have implemented a prototype of our analysis in COSTA, a COSt and Termination Analyser for Java bytecode. Experiments on typical examples from the literature demonstrate that COSTA is able to handle programs with non-linear arithmetics that could not be handled before.

The rest of this paper is organized as follows: Section 2 briefly describes the intermediate language on which we develop our analysis (Java bytecode programs are automatically translated to this language); Section 3 motivates the techniques we use for handling non-linear arithmetic operations; Section 4 describes the different components of our value analysis; Section 5 presents a preliminary experimental evaluation using COSTA; and, finally, we conclude in Section 6.

2 A Simple Imperative Intermediate Language

We present our analysis on a simple *rule-based* imperative language [1] which is similar in nature to other representations of bytecode [23,16]. For simplicity, we consider a subset of the language presented in [1], which deals only with methods and arithmetic operations over integers. In the implementation we handle full sequential Java bytecode. A *rule-based program* P consists of a set of *procedures*. A procedure p with k input arguments $\bar{x} = x_1, \ldots, x_k$ and m output arguments $\bar{y} = y_1, \ldots, y_m$ is defined by one or more *guarded rules*. Rules adhere to this grammar:

$$
\begin{array}{rl}
\mathit{rule} ::= & p(\bar{x}, \bar{y}) \ \leftarrow g, b_1, \ldots, b_n \\
g ::= & \mathit{true} \mid e_1 \ \mathit{op} \ e_2 \mid g_1 \wedge g_2 \\
b ::= & x{:=}e \mid x{:=}e - e \mid x{:=}e + e \mid q(\bar{x}, \bar{y}) \\
& x{:=}e * e \mid x{:=}e \ / \ e \mid x{:=}e \ \texttt{rem} \ e \\
& x{:=}e \ \otimes \ e \mid x{:=}e \oplus e \mid x{:=}e \triangleright e \mid x{:=}e \triangleleft e \\
e ::= & x \mid n \\
\mathit{op} ::= & >|<|\leq|\geq|=
\end{array}
$$

```
int m(int x,int b){
  int y=1;
  int z=0;
  if (b>1) {
    while (y<x){
      z=z+1;
      y=y*b;
    }
  }
  return z;
}
```

$m(\langle x, b\rangle, \langle r\rangle) \leftarrow$
  $y{:=}1,$
  $z{:=}0,$
  $m_1(\langle x, b, y, z\rangle, \langle r\rangle).$
$m_1(\langle x, b, y, z\rangle, \langle r\rangle) \leftarrow$
  $b \leq 1,$
  $r{:=}z.$
$m_1(\langle x, b, y, z\rangle, \langle r\rangle) \leftarrow$
  $b > 1,$
  $m_2(\langle x, b, y, z\rangle, \langle y_1, z_1\rangle),$
  $r{:=}z_1.$

$m_2(\langle x, b, y, z\rangle, \langle y, z\rangle) \leftarrow$
  $y \geq x.$
$m_2(\langle x, b, y, z\rangle, \langle y_2, z_2\rangle) \leftarrow$
  $y < x,$
  $z_1{:=}z + 1,$
  ① $y_1{:=}y * b,$
  $m_2(\langle x, b, y_1, z_1\rangle, \langle y_2, z_2\rangle).$

Fig. 1. A Java program and its intermediate representation. Method m computes $\lceil \log_b(x) \rceil$.

where $p(\bar{x}, \bar{y})$ is the *head* of the rule; g its guard, which specifies conditions for the rule to be applicable; $b_1, \ldots, b_n$ the body of the rule; n an integer; x and y variables and $q(\bar{x}, \bar{y})$ a procedure call by value. The arithmetic operations / and `rem` refer respectively to integer division and remainder. They have the semantics of the bytecode instructions `idiv` and `irem` [17]. Operations $\otimes$, $\oplus$, $\triangleleft$ and $\triangleright$ refer respectively to bitwise AND, bitwise OR, left shift and right shift. They have the semantics of the bytecode instructions `iand`, `ior`, `ishl`, and `ishr` [17]. We ignore the overflow behavior of these instruction, supporting them is left for future work.

The key features of this language which facilitate the formalization of the analysis are: (1) *recursion* is the only iterative mechanism, (2) *guards* are the only form of conditional, (3) there is no operand stack, and (4) rules may have *multiple output* parameters which is useful for our transformation. The translation from Java bytecode programs to rule-based programs is performed in two steps. First, a control flow graph (CFG) is built. Second, a *procedure* is defined for each basic block in the CFG and the operand stack is *flattened* by considering its elements as additional local variables. The execution of rule-based programs mimics standard bytecode [17]. Multiple output arguments in procedures come from the extraction of loops into separated procedure (see Example 2.1). For simplicity, we assume that each rule in the program is given in static single assignment (SSA) form [5].

Example 2.1 Figure 1 depicts the Java code (left) and the corresponding intermediate representation (right) of our running example. Note that our analysis starts from the bytecode, the Java code is shown here just for clarity. Procedure m is defined by one rule, it receives x and b as input, and returns r as output, i.e., r corresponds to the return value of the Java method. Rule m corresponds to the first two instructions of the Java method, it initializes local variables y and z, and then passes the control to m_1. Procedure m_1 corresponds to the **if** statement, and is defined by two mutually exclusive rules. The first one is applied when $b \leq 1$, and simply returns the value of z in the output variable r. The second one is applied when $b > 1$, it calls procedure m_2 (the loop), and upon exit from m_2 it returns the

$$
\begin{array}{l}
m(\langle x, b\rangle, \langle r\rangle) \leftarrow \\
\quad \{y = 1\}, \\
\quad \{z = 0\}, \\
\quad m_1(\langle x, b, y, z\rangle, \langle r\rangle). \\
m_1(\langle x, b, y, z\rangle, \langle r\rangle) \leftarrow \\
\quad \{b \leq 1\}, \\
\quad \{r = z\}. \\
m_1(\langle x, b, y, z\rangle, \langle r\rangle) \leftarrow \\
\quad \{b > 1\}, \\
\quad m_2(\langle x, b, y, z\rangle, \langle y_1, z_1\rangle), \\
\quad \{r = z_1\}.
\end{array}
\qquad
\begin{array}{l}
m_2(\langle x, b, y, z\rangle, \langle y, z\rangle) \leftarrow \\
\quad \{y \geq x\}. \\
m_2(\langle x, b, y, z\rangle, \langle y_2, z_2\rangle) \leftarrow \\
\quad \{y < x\}, \\
\quad \{z_1 = z + 1\}, \\
① \quad \{y_1 = \top\}, \\
\quad m_2(\langle x, b, y_1, z_1\rangle, \langle y_2, z_2\rangle).
\end{array}
$$

Fig. 2. Abstract compilation of the program of Figure 1

value of z_1 in the output variable r. Note that z_1 refers to the value of z upon exit from procedure m_2 (the loop), it is generated by the SSA transformation. Procedure m_2 corresponds to the **while** loop, and is defined by two mutually exclusive rules. The first one is applied when the loop condition is evaluated to *false*, and the second one when it is evaluated to *true*. Note that m_2 has two output variables, they correspond to the values of y and z upon exit from the loop.

3 Motivating Example

Proving that the program of Figure 1 terminates, or inferring lower and upper bounds on its resource consumption (e.g., number of execution steps), requires bounding the number of iterations that its loop can make. Bounding the number of iterations of a loop is usually done by finding a function f from the program states to a well-founded domain, such that if s and s' are two states that correspond to two consecutive iterations, then $f(s) > f(s')$. Traditionally, this function is called *ranking function* [11]. Note that for termination, it is enough to prove that such function exists, while inferring bounds on the resource consumption requires synthesizing such ranking function. For the program of Figure 1, if the program state is represented by the tuple $\langle x, b, y, z\rangle$, then $f(\langle x, b, y, z\rangle) = \mathtt{nat}(x - y)$, where $\mathtt{nat}(v) = \max(v, 0)$, is a ranking function for the **while** loop. Moreover, this function can be further refined to $f(\langle x, b, y, z\rangle) = \log_2(\mathtt{nat}(x - y) + 1)$, which is more accurate for the sake of inferring bounds on the loop's resource consumption.

In this paper we follow the analysis approach used in [1], which divides the value analysis into several steps: (1) an *abstract compilation* [15] step that generates an abstract version of the program, replacing each instruction by an abstract description (e.g., conjunction of linear constraints) that over-approximates its behavior; (2) a fixpoint computation step that computes an abstract semantics of the program; and (3) in the last, we prove termination or infer bounds on resource consumption using the abstract program of point 1 and the abstract semantics of point 2.

Applying the first step on the program of Figure 1 results in the abstract program

of Figure 2. It can be observed that linear arithmetic instructions are precisely described by their corresponding abstract versions. For example, z_1:=z + 1 updates z_1 to hold the value of z + 1, and its corresponding abstract version $\{z_1 = z + 1\}$ is a denotation which states that the value of z_1 is equal to the value of z plus 1. However, in the case of non-linear arithmetic instructions, the abstract description often loses valuable information. This is the case of the instruction y_1:=y ∗ b which is annotated with ① in both Figures 1 and 2. While the instruction updates y_1 to hold the value of y ∗ b, its abstract description $\{y_1 = \top\}$ states that y_1 can take any value. Here $\top$ is interpreted as any integer value. This makes it impossible to bound the number of iterations of the loop, since in the abstract program the function $f(\langle x, b, y, z\rangle) = \texttt{nat}(x - y)$ does not decrease in each two consecutive iterations.

Without any knowledge on the values of y and b, the constraint $\{y_1 = \top\}$ is indeed the best description for y_1:=y∗b when only conjunctions of linear constraints are allowed. However, in the program of Figure 1 it is guaranteed that the value of y is positive and that of b is greater than 1. Using this context information the abstraction of y_1:=y ∗ b can be improved to $\{y_1 \geq 2 * y\}$, which in turn allows synthesizing the ranking function $f(\langle x, b, y, z\rangle) = \texttt{nat}(x - y)$ and its refinement $f(\langle x, b, y, z\rangle) = \log_2(\texttt{nat}(x - y) + 1)$. This suggests that the abstract compilation can benefit from context information when only conjunctions of linear constraints are allowed. However, the essence of abstract compilation is to use only syntactic information, and clearly context information cannot be obtained always by syntactic analysis of the program.

One way to solve the loss of precision when abstracting non-linear arithmetic instructions is to allow the use of disjunctions of linear constraints. For example, the instruction y_1:=y ∗ b could be abstracted to $\varphi_1 \vee \cdots \vee \varphi_n$ where each φ_i is a conjunction of linear constraints that describes a possible scenario. E.g., we could have $\varphi_j = \{y \geq 1, b \geq 2, y_1 \geq 2 * b\}$ in order to handle the case in which $y \geq 1$ and $b \geq 2$. Then, during the fixpoint computation, when the context becomes available, the appropriate φ_i will be automatically selected. However, for efficiency reasons, we restrict our value analysis to use only conjunctions of linear constraints. In order to avoid the use of disjunctive constraints, similarly to [21], we follow an approach that encodes the disjunctive information into the (abstract) program itself. For example, the second rule of m_2 would be abstracted to:

$$
\begin{array}{ll}
m_2(\langle x, b, y, z\rangle, \langle y_2, z_2\rangle) \leftarrow & op_*(\langle a, b\rangle, \langle c\rangle) \leftarrow \{a = 0, c = 0\}. \\
\quad \{y < x\}, & op_*(\langle a, b\rangle, \langle c\rangle) \leftarrow \{a = 1, c = b\}. \\
\quad \{z_1 = z + 1\}, & \qquad\qquad \vdots \\
\textcircled{1}\ \ op_*(\langle y, b\rangle, \langle y_1\rangle), & op_*(\langle a, b\rangle, \langle c\rangle) \leftarrow \{a \geq 2, b \geq 2, c \geq 2 * a\}. \\
\quad m_2(\langle x, b, y_1, z_1\rangle, \langle y_2, z_2\rangle). &
\end{array}
$$

Here, the instruction y_1:=y ∗ b was abstracted to $op_*(\langle y, b\rangle, \langle y_1\rangle)$ which is a call to an auxiliary abstract rule that defines possible abstract scenarios for different inputs. During the fixpoint computation, since op_* is called in a context in which $y \geq 1$ and $b \geq 2$, only the second and last rules of op_* will be selected. Then, these

two rules propagate the constraint $y_1 \geq 2*y$ back, which is required for synthesizing the expected ranking functions, without using disjunctive abstract domains.

4 Value Analysis

In this section we describe the value analysis of COSTA, which is based on the ideas presented in Section 3. The analysis receives as input a program in the intermediate language and a set of initial entries, and, for each (abstract) procedure $p(\bar{x}, \bar{y})$ it infers: (1) A pre-condition (over $\bar{x}$) that holds whenever p is called; and (2) a post-condition (over $\bar{x}$ and $\bar{y}$) that holds upon exit from p. The pre- and post-conditions are conjunction of linear constraints over the domain of Polyhedra [10]. Later, they can be composed in order to obtain invariants for some program points of interest.

In Section 4.1 we describe the abstract compilation step which translates the program P into an abstract version P^α. In Section 4.2 we describe a standard fixpoint algorithm that is used to infer the pre- and post-conditions. Finally, in Section 4.3 we explain how this information is used for bounding the number of iterations of the program's loops.

4.1 Abstract Compilation

This section describes how to transform a given program P into an abstract program P^α. In the implementation, we support also the abstraction of data-structures using the path-length measure [22] (the depth of a data-structure) and the abstraction of arrays to their length. However, in this paper we omit these features since they do not benefit from the techniques we use for abstracting non-linear arithmetic operations. Given a rule $\mathrm{r} \equiv \mathrm{p}(\bar{\mathrm{x}}, \bar{\mathrm{y}}) \leftarrow \mathrm{g}, \mathrm{b}_1, \ldots, \mathrm{b}_\mathrm{n}$, the abstract compilation of r is $r^\alpha \equiv p(\bar{x}, \bar{y}) \leftarrow g^\alpha, b_1^\alpha, \ldots, b_n^\alpha$, where:

(i) the abstract guard g^α is equal to the (linear) guard g;

(ii) if $\mathrm{b_i} \equiv \mathrm{q}(\bar{\mathrm{z}}, \bar{\mathrm{w}})$, then $b_i^\alpha \equiv q(\bar{z}, \bar{w})$;

(iii) if $\mathrm{b_i} \equiv \mathrm{x{:=}e_1 \diamond e_2}$ and $\diamond \in \{+, -\}$, then $b_i^\alpha \equiv \{x = e_1 \diamond e_2\}$; and

(iv) if $\mathrm{b_i} \equiv \mathrm{x{:=}e_1 \diamond e_2}$ and $\diamond \notin \{+, -\}$, then $b_i^\alpha \equiv op_\diamond(\langle e_1, e_2\rangle, \langle x\rangle)$

Then, $P^\alpha = \{r^\alpha \mid \mathrm{r} \in P\}$. Note that we use the same names for constraint variables as those of the program variables (but in italic font for clarity). This is possible since we have assumed that the rules of P are given in SSA form. In the above abstraction, linear guards (point i) and linear arithmetic instructions (point iii) are simply replaced by a corresponding constraint that accurately model their behavior. Note that $\mathrm{x{:=}e_1 \diamond e_2}$ is an assignment while $\{x = e_1 \diamond e_2\}$ is an equality constraint. In point ii, calls to procedures are simply replaced by calls to abstract procedures. In what follows we explain the handling of non-linear arithmetic (point iv).

If the elements of the underlying abstract domain consist only in conjunctions of linear constraints, then non-linear operations are typically abstracted to $\top$. As we have seen in Section 3, this results in a significant loss of precision that prevents bounding the loop's iterations. A well-know solution is to use disjunctions

of linear constraints which allow splitting the input domain into special cases that can be abstracted in a more accurate way. This can be done by directly using disjunctive abstract domains, however, this comes on the price of performance overhead. The solution we use in our implementation, inspired by [21], is to encode the disjunctions in the (abstract) program itself, without the need for using disjunctive abstract domains. In practice, this amounts to abstracting the non-linear arithmetic instruction x:=e$_1\diamond$e$_2$ into a call $op_\diamond(\langle e_1, e_2\rangle, \langle x\rangle)$ to an auxiliary abstract procedure $op_\diamond$, which is defined by several rules that cover all possible inputs and simulate the corresponding disjunction. The rules of $op_\diamond$ are designed by partitioning its input domain and, for each input class, define the strongest possible post-condition. Clearly, the more partitions there are, the more precise are the post-conditions, but the more expensive is the analysis too. Therefore, when designing the rules of $op_\diamond$ this performance and precision trade-off should be taken into account. For the purposes of termination and resource usage analyzes, the partitioning of the input domain aims at propagating accurate information about constancy, equality and progression (e.g, multiplication by a constant), with the least possible number of rules. In what follows, we explain the auxiliary abstract procedures associated to the non-linear arithmetic operations of our language.

Integer division. The auxiliary abstract rule $op_{\texttt{rem}}$ and $op_{/}$ are defined in terms of $op_{\texttt{dr}}$ which stands for $x = y * q + r$:

$$
\begin{array}{|l|}
\hline
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{x = 0, q = 0, r = 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{y = 1, q = x, r = 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{y = -1, q = -x, r = 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{x = y, q = 1, r = 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{x = -y, q = -1, r = 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{x > y > 1, 0 < q \leq \frac{x}{2}, 0 \leq r < y\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{-x > y > 1, \frac{x}{2} \leq q < 0, -y < r \leq 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{x > -y > 1, -\frac{x}{2} \leq q < 0, 0 \leq r < -y\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{-x > -y > 1, 0 < q \leq -\frac{x}{2}, y < r \leq 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{|y| > |x|, q = 0, r = x\}. \\
\hline
op_{/}(\langle x, y\rangle, \langle q\rangle) \quad \leftarrow op_{\texttt{dr}}(\langle x, y\rangle, \langle q, \_\rangle). \\
op_{\texttt{rem}}(\langle x, y\rangle, \langle r\rangle) \leftarrow op_{\texttt{dr}}(\langle x, y\rangle, \langle \_, r\rangle). \\
\hline
\end{array}
$$

Note that, in practice, abstract rules that involve $|\cdot|$ are folded into several cases. The sixth rule, for example, states that if $x > y > 1$ then x/y is a positive number smaller than or equal to $\frac{x}{2}$, and x rem y is a non-negative number smaller than y. This rule is also essential for synthesizing logarithmic ranking functions, when the input value is reduced at least by half in every iteration. Note that we ignore the special cases when $x =$ MIN_VALUE and $y = -1$, since it is a kind of overflow behavior.

Multiplication. The auxiliary abstract procedure op_* is defined as follows:

$$
\begin{array}{l}
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow \{x = 0, z = 0\}. \\
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow \{x = 1, z = y\}. \\
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow \{x = -1, z = -y\}. \\
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow \{x \geq 2, y \geq 2, z \geq 2 * x, z \geq 2 * y\}. \\
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow \{x \leq -2, y \geq 2, z \leq 2 * x, z \leq -2 * y\}. \\
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow \{x \leq -2, y \leq -2, z \geq -2 * x, z \geq -2 * y\}.
\end{array}
$$

We have omitted those rules that can be obtained by swapping the arguments x and y. In this abstraction, we distinguish the cases in which $x = 0$ (constancy), $x = \pm 1$ (equality) and those in which $|x| > 1$ and $|y| > 1$ (progress). Note that, for example, the post-condition $z \geq 2 * x$ is essential for finding a logarithmic ranking function for loops like that of Figure 2. For example, it is not be possible to synthesize such ranking function if we use a weaker, yet sound, post-condition $z > x$.

The bitwise $\otimes$ and $\oplus$. The auxiliary abstract rules $op_\otimes$ and $op_\oplus$ are defined in terms of op_{ao} as follows:

$$
\begin{array}{l}
op_{\mathsf{ao}}(\langle x, y\rangle, \langle a, o\rangle) \leftarrow \{x = 0, a = 0, o = y\}. \\
op_{\mathsf{ao}}(\langle x, y\rangle, \langle a, o\rangle) \leftarrow \{x = -1, a = y, o = -1\}. \\
op_{\mathsf{ao}}(\langle x, y\rangle, \langle a, o\rangle) \leftarrow \{x = y, a = x, o = x\}. \\
op_{\mathsf{ao}}(\langle x, y\rangle, \langle a, o\rangle) \leftarrow \{x > y > 0, 0 \leq a \leq y, o \geq x\}. \\
op_{\mathsf{ao}}(\langle x, y\rangle, \langle a, o\rangle) \leftarrow \{x > 0, y < -1, 0 \leq a \leq x, y \leq o \leq -1\}. \\
op_{\mathsf{ao}}(\langle x, y\rangle, \langle a, o\rangle) \leftarrow \{x < y < -1, a \leq x, y \leq o \leq -1\}. \\
\hline
op_\otimes(\langle x, y\rangle, \langle a\rangle) \leftarrow op_{\mathsf{ao}}(\langle x, y\rangle, \langle a, \_\rangle). \\
op_\oplus(\langle x, y\rangle, \langle o\rangle) \leftarrow op_{\mathsf{ao}}(\langle x, y\rangle, \langle \_, o\rangle).
\end{array}
$$

Since these operations are commutative we omit rules derivable by swapping the input arguments. The first two rules describe the cases $x = 0$ and $x = -1$, i.e., vectors in which all bits are respectively 0 or 1. The third rule handles the case $x = y$. The rest of rules are based on that the result of $x \otimes y$ has less 1-bits than either x or y, whereas the result of $x \oplus y$ has more 1-bits than either x or y.

Shift left and right. Although shift operations in Java bytecode accept any integer value as the shift operand, the number of shifted positions is determined only by the five least significant bits, i.e., it is a value between 0 and $2^5 - 1$ (for type `long` it is determined by the six least significant bits). For the shift left operation $\lhd$, the auxiliary abstract procedure $op_\lhd$ is defined as follows:

$$
\begin{array}{l}
op_\lhd(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x = 0, z = 0\}. \\
op_\lhd(\langle x, s\rangle, \langle z\rangle) \leftarrow \{s = 0, z = x\}. \\
op_\lhd(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x > 0, 0 < |s| < 2^5, z \geq 2x\}. \\
op_\lhd(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x < 0, 0 < |s| < 2^5, z \leq 2x\}. \\
op_\lhd(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x > 0, |s| \geq 2^5, z \geq x\}. \\
op_\lhd(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x < 0, |s| \geq 2^5, z \leq x\}.
\end{array}
$$

The above rules provide an accurate post-condition when the shift operand s satisfies $0 \leq |s| < 2^5$. In the last two abstract rules, the post-conditions are respectively $z \geq x$ and $z \leq x$ since we cannot observe the value of the first five bits of s when $|s| \geq 2^5$. Similarly, for the shift right operation $\triangleright$, the auxiliary abstract rule $op_{\triangleright}$ is defined as follows:

$$
\begin{array}{|l|}
\hline
op_{\triangleright}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x = 0, z = 0\}. \\
op_{\triangleright}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x = -1, z = -1\}. \\
op_{\triangleright}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{s = 0, z = x\}. \\
op_{\triangleright}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x > 0, 0 < |s| < 2^5, x > z, x \geq 2z, z \geq 0\}. \\
op_{\triangleright}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x < -1, 0 < |s| < 2^5, x - 1 \leq 2z, z < 0\}. \\
op_{\triangleleft}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x > 0, |s| \geq 2^5, 0 \leq z \leq x\} \\
op_{\triangleleft}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x < 0, |s| \geq 2^5, x \leq z \leq -1\}\}. \\
\hline
\end{array}
$$

Note that when the program includes several non-linear instructions for the same operations, then it might be useful to generate different auxiliary abstract procedures for them, e.g, op^1_*, op^2_*, etc. This is required mainly when the calling contexts of these instructions are disjoint, and therefore separating their auxiliary abstract procedures avoids merging the calling contexts, which usually results in a loss of precision. In addition, non-linear arithmetic instructions that do not affect the termination of the program can be abstracted as before, i.e., to $\{x = \top\}$, and thus avoid the performance overhead caused by unnecessary auxiliary abstract procedures. These instructions can be identified using dependency analysis, similar to what have been done in [4] for identifying program variables that affect termination.

4.2 Fixpoint algorithm

Algorithm 1 implements the value analysis using a top-down strategy in the style of [7]. It receives as input an abstract program P^α and a set of initial pre-conditions E, and computes pre- and post-conditions for each procedure in P (stored in tables PRE and POST respectively). The meaning of a pre-condition $\mathsf{PRE}[q(\bar{x})] \equiv \varphi$, is that φ holds when calling q, and of a post-condition $\mathsf{POST}[q(\bar{x}, \bar{y})] \equiv \varphi$ is that φ holds upon exit from q.

Procedure FIXPOINT initializes the event queue $\mathcal{Q}$ to $\emptyset$ (L2), initializes the elements of tables PRE and POST to *false* (L4 and L5), processes the initial pre-conditions E by calling ADD_PRE for each one (L6) which in turn adds the corresponding event to $\mathcal{Q}$, and then in the while loop it processes the events of $\mathcal{Q}$ until no more events are available. In each iteration, an event q (a procedure name) is removed from $\mathcal{Q}$ (L8) and processed as follows: the current pre-condition ψ of q is retrieved (L9), each of the rules of q is evaluated in order to generate a post-condition for that specific rule w.r.t. ψ (L11), all post-conditions are joint into a single element δ (using the least upper-bound $\sqcup$ of the underlying abstract domain), and finally δ is added as a post-condition for q by calling ADD_POST. Note that the call to ADD_POST might add more events to $\mathcal{Q}$. The evaluation of a rule (procedure EVALUATE) w.r.t. a pre-condition ψ processes each b^α_i in the rule's body B

Algorithm 1 The fixpoint algorithm

```
1:  procedure FIXPOINT(P^α, E)
2:      Q = ∅;
3:      for all q(x̄, ȳ) ∈ P do
4:          PRE[q(x̄)] = false;
5:          POST[q(x̄, ȳ)] = false;
6:      for all ⟨p(x̄), φ⟩ ∈ E do ADD_PRE(p(x̄), φ);
7:      while Q.notempty() do
8:          q = Q.poll();
9:          ψ = PRE[q(x̄)];
10:         δ = false;
11:         for all q(x̄, ȳ) ← B^α ∈ P^α do δ = δ ⊔ EVALUATE(q(x̄, ȳ) ← B^α, ψ);
12:         ADD_POST(q(x̄, ȳ), δ);
13: function EVALUATE(q(x̄, ȳ) ← B^α, ψ)
14:     for all b_i^α ∈ B^α do
15:         if b_i^α ≡ q'(w̄, z̄) then
16:             ADD_PRE(q'(w̄), ∃̄w̄.ψ);
17:             ψ = ψ ⊓ POST[q'(w̄, z̄)];
18:         else ψ = ψ ⊓ b_i^α;
19:     return ∃̄x̄ ∪ ȳ.ψ;
20: procedure ADD_PRE(q(x̄), φ)
21:     ψ = PRE[q(x̄)];
22:     if φ ⊭ ψ then
23:         PRE[q(x̄)] = ψ ⊔ φ;
24:         Q.add(q);
25: procedure ADD_POST(q(x̄, ȳ), φ)
26:     δ = POST[q(x̄, ȳ)];
27:     if δ ⊭ φ then
28:         POST[q(x̄, ȳ)] = δ ⊔ φ;
29:         for all p ∈ P do
30:             if p calls q then Q.add(p);
```

as follows: if b_i^α is a call $q'(\bar{w}, \bar{z})$, then it registers the corresponding pre-condition by calling ADD_PRE (L16) and adds the current post-condition of q to ψ (L17); otherwise, b_i^α is a constraint and it simply adds it to ψ (L18).

Procedure ADD_PRE adds a new pre-condition for q if it does not imply the current one, and adds the corresponding event to $\mathcal{Q}$. Procedure ADD_POST adds a new post-condition for q if it does not imply the current one, and adds events for all procedures that call q since they might have to be re-analyzed. Note that both procedures use the least upper bound $\sqcup$ of the underlying abstract domain in order to join the new pre- or post-conditions with the current one. Note also that since we use abstract domains with infinite ascending chains, in practice, these procedures incorporate a widening operator in order to ensure termination.

Example 4.1 Consider again the abstract program of Figure 2, where the second abstract rule of m_2 is replaced by

$$m_2(\langle x,b,y,z\rangle,\langle y_2,z_2\rangle) \leftarrow \\ \{y < x\},\ \{z_1 = z+1\},\ op_*(\langle b,y\rangle,\langle y_1\rangle),\ m_2(\langle x,b,y_1,z_1\rangle,\langle y_2,z_2\rangle).$$

and the initial set of entries $E = \{\langle m(\langle x,b\rangle), true\rangle\}$. Then, the fixpoint algorithm infers $\mathsf{PRE}[m_2(\langle x,b,y,z\rangle)] = \{z \geq 0, y \geq 1, b \geq 2\}$, $\mathsf{PRE}[op_*(\langle b,y\rangle)] = \{b > 1, y \geq 1\}$, and $\mathsf{POST}[op_*(\langle b,y\rangle,\langle y_1\rangle)] = \{y_1 \geq 2 * y\}$.

4.3 Bounding the loops

In this section we describe how the abstract program and the pre- and post-conditions are used in order to bound the program's loops, as done in [1]. Briefly, for each abstract rule $p(\bar{x},\bar{y}) \leftarrow g^\alpha, b_1^\alpha,\ldots,b_n^\alpha \in P^\alpha$, we generate a set of transitions

$$\left\{ \langle p(\bar{x}) \rightarrow q(\bar{w}), \bar{\exists}\bar{x}\cup\bar{w}.\varphi\rangle \;\middle|\; \begin{array}{l} i \in [1,\ldots,n],\ b_i^\alpha = q(\bar{w},\bar{z}), \\ \varphi = \ \mathsf{PRE}[q(\bar{x})] \wedge g^\alpha \wedge \phi(b_1^\alpha)\cdots\wedge\phi(b_{i-1}^\alpha) \end{array} \right\}$$

where $\bar{\exists}\bar{x}\cup\bar{w}.\varphi$ is the projection of φ on the variables $\bar{x}\cup\bar{w}$; $\phi(b_i^\alpha) = b_i^\alpha$ if b_i^α is a constraint; and $\phi(b_i^\alpha) = \mathsf{POST}[b_i^\alpha]$ if b_i^α is a call. Then, the set of all transitions is passed to, for example, the tool of [2], which in turn infers ranking functions for the corresponding loops.

Example 4.2 Using the abstract rule and the pre- and post-conditions of Example 4.1, we generate the transition relation

$$\langle m_2(\langle x,b,y,z\rangle) \rightarrow m_2(\langle x,b,y_1,z_1\rangle), \varphi\rangle$$

where $\varphi = \{z \geq 0, y \geq 1, b \geq 2, x < y, z_1 = z+1, y_1 \geq 2*y\}$. Then, the solver of [2] infers the expected ranking functions as explained in Section 3.

5 Experimental Evaluation

We have implemented, in the context of COSTA [3], a prototype of the value analysis described in Section 4. We have performed some experiments on typical examples from the literature that use non-linear and bit arithmetic operations. The benchmarks are available at `http://costa.ls.fi.upm.es/papers/bytecode2011`. Unfortunately, the implementation cannot be tried out via COSTA's web-interface since it has not been integrated in the main branch yet.

COSTA, with the new value analysis, was able to prove termination of all benchmarks. Note that without this value analysis COSTA could not handle any of these benchmarks. We have also analyzed the benchmarks using other termination analyzers for Java bytecode. Julia [1] [22] was not able to prove termination of any of

[1] using the online version `http://julia.scienze.univr.it/`

these benchmarks. AProVE [2] [12] could not prove termination of programs with bit arithmetic operations, but could handle programs with non-linear arithmetic operations such as multiplication and integer division, except for the program of Figure 1 for which it could not complete the proof in a time limit of 5 minutes. In what follows we explain the results of our analysis on some of the benchmarks.

Example 5.1 We start with an example borrowed from [9]:

```
void and(int x){
  while( x > 0)
    x = x & x-1;
}
```

$$
\begin{array}{l}
and(\langle x\rangle,\langle\rangle) \leftarrow and_1(\langle x\rangle,\langle\rangle).\\
and_1(\langle x\rangle,\langle\rangle) \leftarrow \{x \leq 0\}.\\
and_1(\langle x\rangle,\langle\rangle) \leftarrow \{x > 0\},\\
\quad \{y = x-1\},\\
\quad op_{\otimes}(\langle x,y\rangle,\langle x_1\rangle),\\
\quad and_1(\langle x_1\rangle,\langle\rangle).
\end{array}
$$

The code on the right is the abstract compilation of the corresponding intermediate representation of the Java method. In order to bound the number of iterations of the **while** loop, it is essential to infer that the value of x decreases in each iteration. This cannot be guaranteed when considering the instruction x=x & x−1 separately, since, for example, it does not decrease when x=0. Our analysis infers the pre-condition $\mathsf{PRE}[op_{\otimes}(x,y)] = \{y = x-1, x > 0\}$, i.e., the context $x > 0$ is available when calling $op_{\otimes}$, which in turn makes it possible to infer the post-condition $\mathsf{POST}[op_{\otimes}(\langle x,y\rangle,\langle x_1\rangle)] = \{y = x-1, x > 0, 0 \leq x_1 \leq x-1\}$. Using this information we generate the transition $\langle and_1(\langle x\rangle) \rightarrow and_1(\langle x_1\rangle), \{x > 0, 0 \leq x_1 \leq x-1\}\rangle$ for which we synthesize the ranking function $f(\langle x\rangle) = \mathtt{nat}(x)$.

Example 5.2 The next example implements the Euclidean algorithm for computing the greatest common divisor of two natural numbers. It is taken from the Java bytecode termination competition database [3]:

```
int gcd(int a, int b){
  int tmp;
  while(b>0 && a>0){
    tmp = b;
    b = a % b;
    a = tmp;
  }
  return a;
}
```

$$
\begin{array}{l}
gcd(\langle a,b\rangle,\langle r\rangle) \leftarrow gcd_1(\langle a,b\rangle,\langle r\rangle).\\
gcd_1(\langle a,b\rangle,\langle a\rangle) \leftarrow \{a \leq 0\}.\\
gcd_1(\langle a,b\rangle,\langle a\rangle) \leftarrow \{b \leq 0\}.\\
gcd_1(\langle a,b\rangle,\langle r\rangle) \leftarrow\\
\quad \{a > 0, b > 0\},\\
\quad \{tmp = b\},\\
\quad op_{\mathtt{rem}}(\langle a,b\rangle,\langle b_1\rangle),\\
\quad \{a_1 = tmp\},\\
\quad gcd_1(\langle a_1,b_1\rangle,\langle r\rangle).
\end{array}
$$

COSTA was not able to prove termination of this program in the competition of July 2010, mainly because it ignores the calling context when abstracting b=a % b, and therefore it cannot infer that b decreases. Our analysis infers the pre-condition $\mathsf{PRE}[op_{\mathtt{rem}}(\langle a,b\rangle)] = \{a > 0, b > 0\}$, which in turn makes it possible to infer the post-

[2] using the online version http://aprove.informatik.rwth-aachen.de/

[3] http://termcomp.uibk.ac.at

condition $\mathsf{POST}[op_{\texttt{rem}}(\langle a, b\rangle, \langle b_1\rangle)] = \{a > 0, b > 0, b > b_1\}$. Using this information we generate the transition $\langle gcd_1(\langle a, b\rangle) \to gcd_1(\langle a_1, b_2\rangle), \{a > 0, b > 0, b > b_1\}\rangle$ for which we synthesize the ranking function $f(\langle a, b\rangle) = \texttt{nat}(b)$.

Example 5.3 The next example is taken from the method `toString(int i, int radix)` of class `java.lang.Integer`. It is used for writing a number in any numeric base. For simplicity, we have removed code that does not affect the termination, and annotated the loop with a pre-condition that is inferred by our analysis:

```
// { i <= 0, 2 <= radix}
while (i <= -radix) {
    i = i / radix;
}
```

$p(\langle i, radix\rangle, \langle\rangle) \leftarrow \{i > -radix\}.$
$p(\langle i, radix\rangle, \langle\rangle) \leftarrow$
$\quad \{i \leq -radix\},$
$\quad op_{/}(\langle i, radix\rangle, \langle i_1\rangle),$
$\quad p(\langle i_1, radix\rangle, \langle\rangle).$

Due to the pre-condition $\mathsf{PRE}[op_{/}(\langle i, radix\rangle, \langle i_1\rangle)] = \{2 \leq radix, i \leq -radix\}$, our analysis infers the post-condition $\mathsf{POST}[op_{/}(\langle i, radix\rangle, \langle i_1\rangle)] = \{2 \leq radix, i \leq -radix, \frac{i}{2} \leq i_1 < 0\}$. Using this post-condition we generate the transition $\langle p(\langle i, radix\rangle) \to p(\langle i_1, radix\rangle), \{2 \leq radix, i \leq -radix, \frac{i}{2} \leq i_1 < 0\}\rangle$. For this transition we synthesize the ranking function $f(\langle i, radix\rangle) = \log_2(\texttt{nat}(-i) + 1)$.

Example 5.4 The next example is a variation of a loop from the class `Integer` in the method `toUnsignedString(int i, int shift)`, which is used for writing a number in binary, octal or hexadecimal form:

```
// { 1 <= shift <= 4 }
while ( i > 0 ) {
    i >>>= shift;
}
```

$p(\langle i, shift\rangle, \langle\rangle) \leftarrow \{i \leq 0\}.$
$p(\langle i, shift\rangle, \langle\rangle) \leftarrow$
$\quad \{i > 0\},$
$\quad op_{\triangleright}(\langle i, shift\rangle, \langle i_1\rangle),$
$\quad p(\langle i_1, shift\rangle, \langle\rangle).$

Due to the pre-condition $\mathsf{PRE}[op_{\triangleright}(\langle i, shift\rangle] = \{i > 0, 1 \leq shift \leq 4\}$, our analysis infers the post-condition $\mathsf{POST}[op_{\triangleright}(\langle i, shift\rangle, \langle i_1\rangle)] = \{i > 0, 1 \leq shift \leq 4, i \geq 2 * i_1, i_1 \geq 0\}$. Using this postcontidion we generate the transition $\langle p(\langle i, shift\rangle) \to p(\langle i_1, shift\rangle), \{i > 0, 1 \leq shift \leq 4, i \geq 2 * i_1, i_1 \geq 0\}\rangle$, for which we synthesize the ranking function $f(\langle i, shift\rangle) = \log_2(\texttt{nat}(i) + 1)$.

6 Conclusions

In this paper we have described how we handle non-linear arithmetic instructions in the value analysis of COSTA. It is well-know that handling such operations is problematic when the underlying abstract domain allows only the use of conjunctions of linear constraints. It is also well-know that the use of disjunctive abstract domains is a possible solution to this problem, however, on the price of performance overhead. In this paper, instead of using disjunctive abstract domains, we encoded the disjunctive nature of non-linear arithmetic instructions into the abstract program itself. This encoding, when combined with a value analysis that is based on

non-disjunctive abstract domains such as Polyhedra or Octagons, makes it possible to dynamically select the best abstraction depending on the context from which the code that correspond to the encoding was reached. Our experiments demonstrate that COSTA is now able to prove termination and infer bound on resource consumption for programs that it could not handle before. For future work, we plan to improve the scalability of the analyzer, support overflow in arithmetic operations, and support floating point arithmetic. Note that, given the latest developments in the Parma Polyhedra Library [6], supporting overflow and floating point arithmetic is relatively straightforward.

Acknowledgement

This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 *DOVES* project, the HI2008-0153 (Acción Integrada) project, the UCM-BSCH-GR58/08-910502 Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS* project. Diego Alonso is partially supported by the UCM PhD scholarship program.

References

[1] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In Gilles Barthe and Frank de Boer, editors, *IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'08)*, volume 5051 of *Lecture Notes in Computer Science*, pages 2–18, Oslo, Norway, June 2008. Springer-Verlag, Berlin.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *6th International Symposioum on Formal Methods for Components and Objects (FMCO'08)*, number 5382 in Lecture Notes in Computer Science, pages 113–133. Springer, 2007.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Removing Useless Variables in Cost Analysis of Java Bytecode. In *ACM Symposium on Applied Computing (SAC) - Software Verification Track (SV08)*, pages 368–375, Fortaleza, Brasil, March 2008. ACM Press, New York.

[5] A. W. Appel. Ssa is Functional Programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

[6] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy, 2006. Available at `http://www.cs.unipr.it/Publications/`. Also published as `arXiv:cs.MS/0612085`, available from `http://arxiv.org/`.

[7] Michael Codish. Efficient goal directed bottom-up evaluation of logic programs. *J. Log. Program.*, 38(3):355–370, 1999.

[8] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 415–426, Tucson, Arizona, USA, 2006.

[9] Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. Ranking function synthesis for bit-vector relations. In Javier Esparza and Rupak Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 236–250. Springer, 2010.

[10] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *ACM Symposium on Principles of Programming Languages (POPL'78)*. ACM Press, 1978.

[11] R. W. Floyd. Assigning Meanings to Programs. In J.T Schwartz, editor, *Proceedings of Symposium in Applied Mathematics*, volume 19, Mathematical Aspects of Computer Science, pages 19–32. American Mathematical Society, Providence, RI, 1967.

[12] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated Termination Proofs with AProVE. In *Proc. of 15th International Conference on Rewriting Techniques and Applications (RTA'04)*, volume LNCS 3091, pages 210–220. Springer-Verlag, 2004.

[13] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *Symposium on Principles of Programming Languages (POPL'09)*, pages 127–139. ACM, 2009.

[14] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In Benjamin G. Zorn and Alexander Aiken, editors, *PLDI*, pages 292–304. ACM, 2010.

[15] M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, August 1992.

[16] H. Lehner and P. Müller. Formal translation of bytecode into BoogiePL. In *2nd Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode'07)*, Electronic Notes in Theoretical Computer Science, pages 35–50. Elsevier, 2007.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[18] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[19] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 6–86. Elsevier - North Holland, March 2009.

[20] Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. Automated termination analysis of java bytecode by term rewriting. In Christopher Lynch, editor, *RTA*, volume 6 of *LIPIcs*, pages 259–276. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[21] S. Sankaranarayanan, F. Ivancic, I. Shlyakhter, and A. Gupta. Static Analysis in Disjunctive Numerical Domains. In *Static Analysis, 13th International Symposium, (SAS'06)*, volume 4134 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2006.

[22] Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for java bytecode based on path-length. *ACM Trans. Program. Lang. Syst.*, 32(3), 2010.

[23] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In *Proc. of CASCON'99*, pages 125–135. IBM, 1999.

On the Limits of the Classical Approach to Cost Analysis

Diego Esteban Alonso-Blas and Samir Genaim

DSIC, Complutense University of Madrid (UCM), Spain

Abstract. The classical approach to static cost analysis is based on transforming a given program into cost relations and solving them into closed-form upper-bounds. It is known that for some programs, this approach infers upper-bounds that are asymptotically less precise than the actual cost. As yet, it was assumed that this imprecision is due to the way cost relations are solved into upper-bounds. In this paper: (1) we show that this assumption is partially true, and identify the reason due to which cost relations cannot precisely model the cost of such programs; and (2) to overcome this imprecision, we develop a new approach to cost analysis, based on SMT and quantifier elimination. Interestingly, we find a strong relation between our approach and amortised cost analysis.

1 Introduction

Cost analysis (a.k.a. resource usage analysis) aims at *statically* determining the amount of resources required to safely execute a given program, i.e., without running out of resources. By *resource*, we mean any quantitative aspect of the program, such as memory consumption, execution steps, etc. Several cost analysis frameworks are available [2,10,12,14,15,17]. Although different in their underlying theory, all of them usually report the cost of a program as an upper-bound function (UBF for short) such that: when evaluated on (an abstraction of) a given input, the UBF gives an upper-bound on the amount of resources required for safely running the program on that specific input.

Many automatic cost analysis tools are based on the *classical approach* of Wegbreit [22], which we describe using its extension for JAVA bytecode [2]. This analysis is done in three steps: (1) the JAVA program is transformed into an *abstract program*, in which data-structures are abstracted to their sizes, e.g., length of lists, depth of trees, etc.; (2) the abstract program is transformed into a set of *cost relations* (CRs for short), which are a non-deterministic form of *recurrence equations* that define the cost of executing the program in terms of its *input* parameters; and (3) the CRs are solved into UBFs.

This analysis performs well in practice, however, for some classical examples, it infers UBFs that are asymptotically less precise than the actual cost. Clearly, the abstraction at step (1) may involve a loss of precision since it can introduce spurious traces, which do not occur in the original program. This imprecision is out of the scope of this paper. Instead, we focus on the imprecision at steps (2)

A. Miné and D. Schmidt (Eds.): SAS 2012, LNCS 7460, pp. 405–421, 2012.

and (3). As yet, it was *assumed* that this imprecision is due to the way CRs are solved into UBFs in step (3), and that, in principle, it could be overcome using more precise resolution techniques.

The *first contribution* of this paper shows that this assumption is not true, namely, that the cost of some programs cannot be modeled precisely with CRs. This is because CRs are defined only in terms of the input parameters, and thus they fail to capture dependencies between the output of a program and its cost. These dependencies are crucial for programs in which the output of one part is passed as input to another part, and transforming them into CRs introduces spurious scenarios. Any resolution technique that solves CRs into UBFs must cover these spurious scenarios, hence it would fail to obtain precise UBFs.

To eliminate these spurious scenarios, an UBF must be defined in terms of both input and output. Our *second contribution* is a novel cost analysis that uses the this notion of cost. It is based on quantifier elimination and template UBFs. Briefly, it takes a given set of template UBFs, with some unknown parameters, and uses satisfiability modulo theory (SMT) and quantifier elimination to instantiate those parameters, such that the resulting UBFs are safe.

The rest of the paper is organised as follows. Sec. 2 presents our running examples and formally defines the language on which we apply our analysis. Sec. 3 studies the limitations of CRs. Secs. 4 and 5 are the technical core of the paper, in which we develop our cost analysis. Sec. 6 discusses the relation of our analysis to amortised cost analysis. Sec. 7 describes a prototype implementation. Sec. 8 overviews related work, and Finally, Sec. 9 concludes.

2 Motivating Examples and Preliminaries

In this section we describe an *abstract cost rules* (ACR for short) language [2], which we use to formally present our cost analysis. In [2], a JAVA program is *automatically abstracted* to this language. The abstraction guarantees that every *concrete* trace has a corresponding *abstract* one with the same cost, but there might be spurious abstract traces, which do not correspond to concrete ones. Recall that our interest is in analysing ACR programs, the translation from JAVA is out of the scope of this paper. We first explain the language using some examples that we use along the paper. Then, we formally define its syntax, semantics and the concrete notions of cost. As a notation, we refer to line number `n` in a given JAVA (resp. ACR) program by J`n` (resp. A`n`).

Example 1. The JAVA code of the first example is depicted in Fig. 1 (on the left). It implements a Stack data-structure using a linked list whose first element is the top of the stack (field top points to this list). Method main has a loop (J14-19) that in each iteration invokes method randPop (J15), which in turn pops an arbitrary number of elements (J6-9), and then pushes a new element (J16). Note that coin() at J6 non-deterministically returns *true* or *false*. Each pop operation consumes m resources, as specified by the annotation @acquire(m) at J8, and each push consumes 1 resource (J17). This example is based on a classical example for amortised analysis [9], the only difference is that pop costs m units instead

```
1 class Stack {
2   Node top;
3
4   //@requires m >= 1
5   void randPop(int m) {
6     while( top != null && coin() ) {
7       top=top.next; //pop
8       //@acquire(m)
9     }
10  }
11
12  //@requires m >= 0
13  void main(int m) {
14    while( m > 0 ) {
15      randPop(m);
16      top = new Node('a',top); //push
17      //@acquire(1)
18      m = m−1;
19    }
20  }
21 }
```

```
1  rpop([s, m], [s1]) ←
2    m ≥ 1,
3    s ≥ 0,
4    s1 = s.
5  rpop([s, m], [s1]) ←
6    m ≥ 1,
7    s ≥ 1,
8    acq(m),
9    s2 = s − 1,
10   rpop([s2, m], [s1]).
11
12 main([s, m], [s1]) ←
13   m = 0,
14   s1 = s,
15 main([s, m], [s1]) ←
16   m ≥ 1,
17   rpop([s, m], [s2]),
18   s3 = s2 + 1,
19   acq(1),
20   m1 = m − 1,
21   main([s3, m1], [s1]).
```

Fig. 1. Java code for Stack and its ACR program

of 1, to showcase some unique features of our analysis. These m units can be seen as the cost of executing m iterations of a loop (which we omit).

Fig. 1 (on the right) includes the ACR version of Stack. It has been automatically generated, and simplified for clarity, using the tools of [2]. A1-10 define a procedure *rpop* that corresponds to randPop. It has two input parameters: s is the size of the stack (i.e., the length of list top); and m is the value of variable m. Note that s is an abstraction of top. It also has one output parameter s_1 which corresponds to the size of the stack upon exit from randPop. Procedure *rpop* is defined by means of two rules: the first one (A1-4) corresponds to the case in which we do not enter the loop; and the second one (A5-10) corresponds to executing one iteration and calling *rpop* recursively (A10) for more iterations. The instruction $s_2 = s - 1$ at A9 corresponds to removing an element from the stack (J7). The translation of method main into procedure *main* (A12-20) is done in a similar way. Just note that calling *rpop* (A17) with a stack of size s results in a stack of size s_2, and that $s_3 = s_2 + 1$ at A18 corresponds to J16.

A call $main([s, m], [s_1])$ executes exactly m push operations, and thus, it can execute at most $s + m$ pop operations. Each push costs exactly 1, and each pop at most m. Since m varies from one call to *rpop* to another, then $s{\cdot}m + \frac{1}{2}(m^2 + m)$ is an UBF on the resource consumption of $main([s, m], [s_1])$. The analysis of [2] infers the cubic UBF $m^3 + s{\cdot}m^2 + m$, which is asymptotically less precise.

408 D.E. Alonso-Blas and S. Genaim

```
1 //@requires n>=0
2 void p(int n) {
3   if (n > 0) {
4     m = q(n);
5     //@release(m)
6     p(n−m);
7     //@release(m)
8   }
9 }
```

```
10 //@requires n>=1
11 int q(int n) {
12   int i = n/2;
13   do {
14     A x = new A();
15     B y = new B();
16     //@acquire(2)
17     i−−;
18     // [...]
19   } while(i>0 && coin());
20   return n/2 − i;
21 }
```

```
1  p([n], [ ]) ←
2    n = 0.
3  p([n], [ ]) ←
4    n ≥ 1,
5    q([n], [m]),
6    rel(m),
7    n1 = n − m,
8    p([n1], [ ]),
9    rel(m).
10 l([i], [i1]) ←
11   i ≥ 0,
12   acq(2),
13   i1 = i − 1.
14 l([i], [i1]) ←
15   i ≥ 1,
16   acq(2),
17   i2 = i − 1,
18   l([i2], [i1]).
19 q([n], [m]) ←
20   n ≥ 1
21   i = n/2,
22   l([i], [i1]),
23   m = i − i1.
```

Fig. 2. Java code for the peak, and its ACR program

Example 2. The second example is depicted in Fig. 2. We use it to explain the notion of *peak* resource consumption. Method `q` (J10-21) receives an integer `n`, executes at least 1 and at most `n`/2 iterations of a loop (J13-19), and returns the number of iterations that have been performed. This loop creates 2 objects in each iteration (J14-15). Method `p` executes a loop (using recursion) where in each iteration it calls `q` with the current value of the loop counter `n`, and then performs a recursive call where the loop counter is decremented by `m` (the number of iterations that `q` has performed). The ACR version, depicted in Fig. 2 on the right, its relation to the JAVA code is as in Ex. 1. We skip details and only comment that procedure l (A10-23) corresponds to the while loop (J13-19). Note that the ACR includes explicit resource release instructions (A6 and A9).

A call $p([n], [\,])$ creates exactly $2{\cdot}n$ objects. However, assuming that objects of type `A` (resp. `B`) become unreachable at J5 (resp. J7), then m objects can be garbage collected when reaching J5 (resp. J7). Thus, at any given moment there cannot be more than n reachable objects, which means that a memory for n objects (the peak consumption) is enough for safely executing this program. The analysis of [2,3] infers the UBF $\frac{n\cdot(n+1)}{2}$ which is asymptotically less precise.

In both programs of Exs. 1 and 2, the resource consumption is specified with the annotations $\texttt{acq}(e)$ and $\texttt{rel}(e)$, for acquiring and releasing e resources respectively. It should be clear that we are interested in inferring safe UBFs assuming the given annotations, and not in inferring the annotations.

Syntax. Formally, an ACR program is a set of procedures. A procedure p is defined by a set of rules of the form $p(\bar{x}, \bar{y}) \leftarrow b_1, b_2, \ldots, b_n$ where $\bar{x}$ (resp. $\bar{y}$) is a sequence of input (resp. output) parameters, and each b_i is one of the following instructions: a (linear) constraint φ; a procedure call $q(\bar{w}, \bar{z})$; or a resource consumption instruction $\texttt{acq}(e)$ or $\texttt{rel}(e)$ where e is an arithmetic expression that evaluates to a non-negative value. In the rest of the paper we assume a given program P (to avoid repeating "for a given program P").

$$\textcircled{1}\ \frac{q(\bar{x},\bar{y}) \leftarrow \bar{b}' \in P}{\langle \psi, q(\bar{x},\bar{y}) \cdot \bar{b}\rangle \overset{0}{\rightsquigarrow} \langle \psi, \bar{b}' \cdot \bar{b}\rangle} \qquad \textcircled{2}\ \frac{\psi \wedge \varphi \not\models \mathit{false}}{\langle \psi, \varphi \cdot \bar{b}\rangle \overset{0}{\rightsquigarrow} \langle \psi \wedge \varphi, \bar{b}\rangle}$$

$$\textcircled{3}\ \frac{\mathit{eval}(e,\psi) = v \geq 0}{\langle \psi, \mathtt{acq}(e) \cdot \bar{b}\rangle \overset{v}{\rightsquigarrow} \langle \psi, \bar{b}\rangle} \qquad \textcircled{4}\ \frac{\mathit{eval}(e,\psi) = v \geq 0}{\langle \psi, \mathtt{rel}(e) \cdot \bar{b}\rangle \overset{-v}{\rightsquigarrow} \langle \psi, \bar{b}\rangle}$$

Fig. 3. Semantics of ACR programs

Semantics. A state s takes the form $\langle \psi, \bar{b}\rangle$, where $\bar{b}$ is a sequence of instructions pending for execution, and ψ is a constraint over $\mathit{vars}(\bar{b})$ and possibly other existentially quantified variables. The *store* ψ imposes relations between variables (e.g., $x = 1$, $x > y$). An execution starts from an initial state $\langle \bar{x} = \bar{v}, p(\bar{x},\bar{y})\rangle$, where $\bar{v}$ is a sequence of integers, which is then rewritten according to the rules in Fig. 3. These rules define a transition relation $s_1 \overset{v}{\rightsquigarrow} s_2$, meaning that there is a transition from s_1 to s_2 that consumes v resources. Rule ① handles procedure calls, it (non-deterministically) selects a rule from P that matches the call, and adds its instructions $\bar{b}'$ to the sequence of pending instructions. Variables in $\bar{b}'$ (except $\bar{x} \cup \bar{y}$) are renamed such that they are different from $\mathit{vars}(\bar{b}) \cup \mathit{vars}(\psi)$. Rule ② handles constraints by adding them to the store, if the resulting state is satisfiable. Rules ③-④ handle resource consumption. They evaluate e to a non-negative value v, and label the corresponding transition with v or $-v$.

The execution stops when no rule is applicable, which happens when the execution reaches (1) a *final state* $\langle \psi', \epsilon\rangle$ where ϵ is the empty sequence; or (2) a *blocking state* $\langle \psi', \varphi \cdot \bar{b}\rangle$ where $\varphi \wedge \psi' \models \mathit{false}$. A trace t is a finite or infinite sequence of states in which there is a valid transition between each pair of consecutive states. Traces that end in a final state and infinite traces are called *complete*. Namely, we exclude traces that end in a blocking state. We write $s_1 \overset{*}{\rightsquigarrow} s_2$ for a finite trace starting from s_1 and ending at s_2.

Definition 1 (trace cost). *Given a finite trace t, its net-cost $\tilde{\tau}(t)$ is the sum of the cost labels on its transitions. Given a complete trace t, its peak-cost $\hat{\tau}(t)$ is defined as* $\mathsf{max}\{\tilde{\tau}(t') \mid t' \text{ is a prefix of } t\}$.

Note that the peak-cost is always non-negative since the empty trace is a prefix of any trace t. However, the net-cost can be also negative. This is because we do not require that resources are acquired before they are released. This is useful for modeling consumer/producer programs, where the produced data can be viewed as resources. Though, we do not address such scenarios in this paper.

Definition 2 (procedure cost). *Given a procedure p with m input and n output parameters, its net-cost $\tilde{\pi}(p)$ and peak-cost $\hat{\pi}(p)$ are defined as*

$$\begin{aligned}\tilde{\pi}(p) &= \{\langle \bar{v}_1, \bar{v}_2, \tilde{\tau}(t)\rangle \mid \bar{v}_1 \in \mathbb{Z}^m, \bar{v}_2 \in \mathbb{Z}^n, t \equiv \langle \bar{x} = \bar{v}_1, p(\bar{x},\bar{y})\rangle \overset{*}{\rightsquigarrow} \langle \psi, \epsilon\rangle, \bar{y} = \bar{v}_2 \models \psi\}\\ \hat{\pi}(p) &= \{\langle \bar{v}_1, \hat{\tau}(t)\rangle \mid \bar{v}_1 \in \mathbb{Z}^m, t \text{ is a complete trace and starts in } \langle \bar{x} = \bar{v}_1, p(\bar{x},\bar{y})\rangle\}\end{aligned}$$

Intuitively, the net-cost tells what is the balance between the resources that have been acquired and released during the execution of p. Note that it only considers

traces that terminate in a final state. The peak-cost tells what is the maximum amount of resources that a program can hold (i.e., acquired but not released yet) at any given state during the execution. Note that Def. 2 does not consider traces that terminate in a blocking state. This is because they do not correspond to valid traces in the JAVA program, and obtained due to the abstraction.

We say that $\mathcal{C} \geq 0$ resources are enough for safely executing $p(\bar{v}, \bar{y})$ without running out of resources if $\mathcal{C} \geq \mathsf{max}\{c \mid \langle \bar{v}, c\rangle \in \hat{\pi}(p)\}$. Note that for terminating programs that only acquires resources, one could also use $\mathcal{C} \geq \mathsf{max}\{c \mid \langle \bar{v}, \bar{v}', c\rangle \in \tilde{\pi}(p)\}$. This is the case for example of the Stack program. Our main interest is in inferring UBFs on the peak-cost of each procedure, however, this will require inferring first UBFs on the net-cost of each procedure p as we will see later.

3 Shortcomings of the Classical Approach to Cost Analysis

As explained in Sec. 1, the classical approach to cost analysis first transforms a given program into a set of CRs, and then solves these CRs into UBFs. The following CRs are automatically generated by [2] for the Stack program of Fig. 1

(1)	$rpop(s, m) = 0$	$\{m \geq 1 \wedge s \geq 0\}$
(2)	$rpop(s, m) = m + rpop(s_2, m)$	$\{m \geq 1 \wedge s \geq 1 \wedge s_2 = s - 1\}$
(3)	$main(s, m) = 0$	$\{m{=}0 \wedge s \geq 0\}$
(4)	$main(s, m) = 1 + rpop(s, m) + main(s_3, m_1)$	$\{m{\geq}1 \wedge s_3{=}s_2{+}1 \wedge m_1{=}m{-}1 \wedge \underline{s{\geq}s_2{\geq}0}\}$

Eqs. (1)-(2) capture the cost of executing procedure *rpop* on the input s and m, and Eqs. (3)-(4) capture the cost of executing procedure *main* on the input s and m. Eq. (4) states that when $m \geq 1$, the cost of executing $main(s, m)$ is 1 (for the push operation); plus the cost of executing $rpop(s, m)$; plus the cost of executing $main(s_3, m_1)$. The constraints on the right side of each equation define the applicability conditions for that equation (e.g., $m \geq 1$) and relations between its variables (e.g., $s_3 = s_2 + 1$). Note that the above CRs have a similar structure to the corresponding ACR program of Fig. 1.

A fundamental difference between ACRs and CRs is that the latter do not include the output parameters. For example, in Eq. (4), the output parameter s_2 in the call to *rpop* has been removed, and the constraint $s \geq s_2 \geq 0$ (underlined in Eq. (4)) has been added to indicate that, upon exit from *rpop*, the value of s_2 is non-negative and smaller than or equal to s. Note that this is the most precise relation between the input and the output parameters of *rpop*. This information is obtained by value analysis (at the level of the ACR program) that infers relations between the input and the output parameters [6].

CRs can be evaluated (they are similar to a functional program with constraints) to obtain the cost of a corresponding procedure. E.g., $main(v_1, v_2)$ can be evaluated to obtain the cost of executing $main([v_1, v_2], [y])$. Clearly, due to the non-determinism (e.g., in the constraints), the evaluation of $main(v_1, v_2)$ might result in several possible values. Soundness requires that the cost of any trace for $main([v_1, v_2], [y])$ is a possible result for $main(v_1, v_2)$. Nevertheless, the

interest is not in evaluating CRs, since it is like executing the ACR program, but rather in statically computing UBFs that bound their results. For example, the solver of [1] infers the UBF $m^3 + m^2{\cdot}s + m$ for $main(s, m)$. Intuitively, it does this as follows: (a) it infers the maximum number of iterations that *main* can perform, which is m; (b) it infers a worst-case behaviour for all iterations, which is $1 + (s + m){\cdot}m$ since the stack can have at most $s + m$ elements; and (c) it multiplies (a) and (b) to get the above UBF.

It is known that, in practice, cost analysers that are based on CRs fail to obtain the desired UBFs for programs like those in Fig. 1 and 2. Moreover, as yet, it was assumed that this failure is due to (i) the way CRs are solved into UBFs; and (ii) the imprecision in the value analysis which is used to infer input-output relations (as $s \geq s_2 \geq 0$ above). It was also assumed that, in principle, one could develop more sophisticated techniques for solving CRs [4] or use more precise value analysis (e.g., non-linear) that would obtain precise UBFs for such programs. In what follows we show that these assumptions are not true. In particular, that Eqs. (3)-(4) in the above CRs do not model *precisely* the cost of procedure *main*, and thus any sound UBF for *main* would be imprecise.

Let us consider an evaluation of $main(s, m)$ in the above CRs. It is easy to see that, using Eq. (4), we can choose $s_2 = s$ and thus get $main(s, m) = 1 + rpop(s, m) + main(s + 1, m - 1)$. Then, in the same way, we can get $main(s + 1, m - 1) = 1 + rpop(s + 1, m - 1) + main(s + 2, m - 2)$, and so on for each $main(s+i, m-i)$. Thus, an evaluation of $main(s, m)$ admits $\sum_{i=0}^{m-1}(1 + rpop(s + i, m - i))$ as a possible result. Since $rpop(s, m)$ can always evaluate to $s{\cdot}m$, the above sum can be reduced to $u(s, m) = \frac{(m-1)}{6}{\cdot}(m^2 + 3{\cdot}s{\cdot}m + m + 6)$. This means that any UBF $f(s, m)$ for Eqs. (3)-(4) must satisfy $\forall s, m : f(s, m) \geq u(s, m)$, which is asymptotically less precise than the UBF from Ex. 1. Thus, we conclude that the imprecision is not related to how CRs are solved, and not to imprecision in the value analysis since the input-output relation $s \geq s_2 \geq 0$ that we used above is the most precise one.

The actual reason for this imprecision is that, in Eq (4), the value for s_2, i.e., the output of *rpop*, and the cost of $rpop(m, s)$ can be chosen independently. For example, in the original program it is not possible that $s_2 = s$ and that the cost of $rpop(s, m)$ is $s{\cdot}m$, in which case s_2 must be 0. However, in the above CRs this scenario is possible. This relation cannot be captured if the UBFs are defined only in terms of the input parameters, an observation that lead us to the idea of defining UBFs in terms of both input and output parameters.

Example 3. Consider again procedure $rpop([s, m], [s'])$ of Fig. 1. The CRs-based approach infers the UBF $s{\cdot}m$ for *rpop*, which depends only on the input parameters s and m. This indeed is the most precise UBF if only input parameters are allowed, since there exists an execution in which we remove all stack elements. However, if we allow the use of output parameters also, then $(s - s'){\cdot}m$ describes the exact cost of *rpop*: $s - s'$ is the number of elements that have been removed from the stack, and removing each one costs m.

At this point, the use of output parameters to define UBFs might look inappropriate. This is because UBFs are usually used to *statically* estimate the amount

of resource required for safely executing the program. However, requiring information on the output parameters in order to evaluate a given UBF is like actually requiring to execute the program. This is not really the case because of the following two reasons. First, when inferring UBFs on the net-cost, we distinguish between the entry procedure (e.g., *main*), and intermediate procedures (e.g., *rpop*). The UBF for the entry procedure will (almost always) be definable in terms of its input parameters only, however, in order to infer a precise UBF for the entry procedure, we need UBFs for the intermediate procedures in terms of input and output parameters. Second, UBFs on the peak-cost, which are the important ones for safety, will use only input parameters, however, inferring them will make use of net-cost UBFs that depend on input and output parameters.

4 Inference of Net-Cost

In this section we describe our approach for inferring UBFs on the net-cost of the program's procedures, which is based on defining the cost in terms of the input and output parameters. We show that it can infer the precise cost of the Stack example of Fig. 1. In Sec. 5, we extend it to infer UBFs on the peak-cost.

Definition 3 (safe net-cost UBFs). *Let p be a procedure with n input and m output parameters. A function $\tilde{f}_p : \mathbb{Z}^{n+m} \mapsto \mathbb{Q}$ is a safe UBF on the net-cost of p iff for any $\langle \bar{v}_1, \bar{v}_2, c\rangle \in \tilde{\pi}(p)$ it holds $\tilde{f}_p(\bar{v}_1, \bar{v}_2) \geq c$.*

Intuitively, a function $\tilde{f}_p$ is an UBF on the net-cost of p if for any possible execution that starts with input $\bar{v}_1$, terminates in a final state with an output $\bar{v}_2$, and have net-cost c, it holds that $\tilde{f}_p(\bar{v}_1, \bar{v}_2) \geq c$. Clearly, CRs cannot be used to infer such UBFs, since they do not use the output parameters.

In what follows we develop a novel approach for inferring such UBFs that is based on the use of quantifier elimination. We present our approach in two steps: (1) *verification*: in which we are given a set of *candidate* UBFs on the net-cost of each procedure, and our interest is to verify that these functions are safe, i.e., satisfy Def. 3; and (2) *inference*: in which we are given a set of *template* UBFs, and our interest is to instantiate the templates parameters into safe UBFs.

Verification of UBFs on the Net-Cost. Let us start by explaining the basics of the verification step. Assume that we have a procedure p defined by the following single rule

$$p(\bar{x}, \bar{y}) \leftarrow \texttt{acq}(e), q_1(\bar{x}_1, \bar{y}_1), \ldots, q_n(\bar{x}_n, \bar{y}_n)$$

and that we have a set of *safe* UBFs $\tilde{f}_{q_1}, \ldots, \tilde{f}_{q_n}$ on the net-cost of $q_1, \ldots, q_n$. To verify that a given $\tilde{f}_p$ is a safe UBF on the net-cost of p, it is *sufficient* to check that the condition $\tilde{f}_p(\bar{x}, \bar{y}) \geq e + \tilde{f}_{q_1}(\bar{x}_1, \bar{y}_1) + \cdots + \tilde{f}_{q_n}(\bar{x}_n, \bar{y}_n)$ holds for any values of the program variables. Applying this principle to all rules of the program, it is possible to verify the safety of several candidate UBFs simultaneously.

Given a set $\tilde{F}$ of candidate UBFs on the net-cost that includes a function $\tilde{f_p} : \mathbb{Z}^{n+m} \mapsto \mathbb{Q}$ for each procedure $p \in P$, we build a *verification condition* (VC for short) whose validity implies the safety of each $\tilde{f_p} \in \tilde{F}$. The net-cost VC is generated from the program rules as follows.

Definition 4 (Net-cost VC). *Given a set $\tilde{F}$ of candidate UBFs, for each rule $r \equiv p(\bar{x}, \bar{y}) \leftarrow b_1, b_2, \ldots, b_n$, we generate a condition ψ_r as follows:*

1. *let φ be the conjunction of all constraints in r;*
2. *let the net-cost $\tilde{b}$ of an instruction b be defined as follows: if $b \equiv q_i(\bar{x}_i, \bar{y}_i)$ then $\tilde{b} \equiv \tilde{f_q}(\bar{x}_i, \bar{y}_i)$, if $b \equiv \texttt{acq}(e)$ then $\tilde{b} \equiv e$, if $b \equiv \texttt{rel}(e)$ then $\tilde{b} \equiv -e$, and if b is a constraint then $\tilde{b} \equiv 0$;*
3. *let $\psi_r \equiv \forall \bar{w} : \varphi \Rightarrow \tilde{f_p}(\bar{x}, \bar{y}) \geq \tilde{b_1} + \cdots + \tilde{b_n}$ where $\bar{w} = vars(r)$.*

Then, the net-cost VC is defined as $\Psi(\tilde{F}) = \wedge_{r \in P} \psi_r$.

Note that ψ_r is the condition we explained before, but taking into account the constraints φ of the rule r which define the context in which this condition holds.

Example 4. Consider the program in Fig. 1, and let $\tilde{f_r}(s, m, s_1)$ and $\tilde{f_m}(s, m, s_1)$ be candidate UBFs on the net-cost of $rpop([s, m], [s_1])$ and $main([s, m], [s_1])$, respectively. The verification condition for this program w.r.t. $\tilde{F} = \{\tilde{f_r}(s, m, s_1), \tilde{f_m}(s, m, s_1)\}$ is $\Psi(\tilde{F}) = \psi_{r_1} \wedge \psi_{r_2} \wedge \psi_{r_3} \wedge \psi_{r_4}$ where:

$$\psi_{r_1} \equiv \forall \bar{w}_1 : m \geq 1 \wedge s \geq 0 \wedge s_1 = s \Rightarrow \tilde{f_r}(s, m, s_1) \geq 0$$
$$\psi_{r_2} \equiv \forall \bar{w}_2 : m \geq 1 \wedge s \geq 1 \wedge s_2 = s - 1 \Rightarrow \tilde{f_r}(s, m, s_1) \geq m + \tilde{f_r}(s_2, m, s_1)$$
$$\psi_{r_3} \equiv \forall \bar{w}_3 : m = 0 \wedge s_1 = s \wedge s \geq 0 \Rightarrow \tilde{f_m}(s, m, s_1) \geq 0$$
$$\psi_{r_4} \equiv \forall \bar{w}_4 : \begin{array}{l} m \geq 1 \wedge s_3 = s_2 + 1 \wedge \\ m_1 = m - 1 \wedge s \geq 0 \end{array} \Rightarrow \tilde{f_m}(s, m, s_1) \geq \tilde{f_r}(s, m, s_2) + 1 + \tilde{f_m}(s_3, m_1, s_1)$$

The condition ψ_{r_4}, for example, corresponds to the second rule of procedure *main*. It states that $\tilde{f_m}(s, m, s_1)$ is a safe UBF if it is greater than the cost of the call to *rpop*, i.e., $\tilde{f_r}(s, m, s_2)$, plus 1 for the push operation, plus the cost of the recursive call to *main*, i.e, $\tilde{f_m}(s_3, m_1, s_1)$. This condition should hold for any values that satisfy the constraint $m \geq 1 \wedge s_3 = s_2 + 1 \wedge m_1 = m - 1 \wedge s \geq 0$, i.e., in the context of the second rule. Let us consider now the validity of $\Psi(\tilde{F})$ for the following possible concrete definitions of $\tilde{f_r}(s, m, s_1)$ and $\tilde{f_m}(s, m, s_1)$

(a) $\tilde{f_m}(s, m, s_1) = s{\cdot}m + \frac{1}{2}(m^2 + m)$, and $\tilde{f_r}(s, m, s_1) = (s - s_1){\cdot}m$
(b) $\tilde{f_m}(s, m, s_1) = s{\cdot}m + \frac{1}{2}(m^2 + m)$, and $\tilde{f_r}(s, m, s_1) = s{\cdot}m$

Using (a), we get that $\Psi(\tilde{F})$ is a valid formula. Note that here we use the optimal UBFs for *main* and *rpop*. Using (b), we get that $\Psi(\tilde{F})$ is invalid, though both UBF are safe. This is because, in this case, using $s{\cdot}m$ as an UBF for *rpop* is not enough for proving that $s{\cdot}m + \frac{1}{2}(m^2 + m)$ is an UBF for *main*.

Theorem 1. *Given a set $\tilde{F}$ of candidate UBFs, if $\models \Psi(\tilde{F})$ then $\tilde{F}$ is safe.*

Note that checking the validity of $\Psi(\tilde{F})$ is a first order problem that can be solved using SMT solvers (see Sec. 7).

Inference of UBFs on the net-cost. For many applications it is useful to infer the set $\tilde{F}$, instead of verifying the correctness of a given one. This can be formulated as seeking a set $\tilde{F}$ of UBFs for which $\Psi(\tilde{F})$ is valid, which means solving the formula $\exists \tilde{f}_1 \tilde{f}_2 \dots \tilde{f}_k : \Psi(\tilde{F})$. However, this is a second order problem and solving it in general is impractical. A common approach to avoid solving a second order formula is the use of template functions that restrict the form of functions that we are looking for. A template for $\tilde{f}_p(\bar{x}, \bar{y})$ is a function with a fixed structure, defined over the variables $\bar{x} \cup \bar{y}$, and some unknown template parameters.

Example 5. The following are UBF templates for procedure *main* and *rpop*:

1. $\tilde{f}_r(s, m, s_1) = \lambda_1 \cdot s \cdot m + \lambda_2 \cdot s_1 \cdot m + \lambda_3 \cdot s + \lambda_4 \cdot m + \lambda_5 \cdot s_1 + \lambda_0$
2. $\tilde{f}_m(s, m, s_1) = \mu_1 \cdot s \cdot m + \mu_2 \cdot m^2 + \mu_3 \cdot s_1 \cdot m + \mu_4 \cdot s + \mu_5 \cdot m + \mu_6 \cdot s_1 + \mu_0$

The variables $\bar{\lambda}$ and $\bar{\mu}$ are the template parameters.

Assuming that $\tilde{F}$ is a set of candidate UBF templates, and that $\mathcal{P}$ is the set of template parameters, the inference problem is reduced to solving the first order problem $\exists \mathcal{P} : \Psi(\tilde{F})$. This can be solved by combining quantifier elimination and SMT solvers (see Sec. 7). The idea behind UBF templates is that later we will assign values to the template parameters such that the resulting UBFs are safe.

Note that in Ex. 5 we have chosen simple templates just to keep the technical details in the next examples simple. We could also choose a cubic polynomial template, and later try to find an instantiation such that the parameters of the cubic parts are assigned 0 (in order to get the quadratic UBF). In principle, any template UBF can be used as far as it uses arithmetic expressions that are supported by the quantifier elimination procedure (see Sec. 7).

Example 6. Using the templates of Ex. 5 in the VC of Ex. 4, we get a VC $\Psi(\tilde{F})$ in which the template variables $\bar{\lambda} \cup \bar{\mu}$ are free variables. Eliminating the universally quantified variables, we get a formula ξ over $\bar{\lambda} \cup \bar{\mu}$ that is a conjunction of the following equalities and inequalities:

$$\begin{array}{l|l|l|l|l} \lambda_1 \geq 1 & \lambda_2 = -\lambda_1 & \lambda_1 + \lambda_3 \geq 1 & \mu_6 \geq \lambda_5 - \lambda_1 & 2 \cdot \mu_2 \geq \lambda_1 + \lambda_4 \\ \lambda_4 \geq 0 & \mu_1 = \lambda_1 & \lambda_3 + \lambda_5 \geq 0 & \mu_4 = \lambda_1 - \lambda_5 & \mu_5 + \mu_2 \geq \mu_4 + \lambda_0 + \lambda_4 + 1 \\ \mu_0 \geq 0 & \mu_3 = 0 & \lambda_0 + \lambda_4 \geq 0 & \lambda_1 \geq \lambda_3 + \lambda_5 & \end{array}$$

Each model of ξ assigns values to the template parameters $\bar{\lambda}$ and $\bar{\mu}$ such that $\tilde{f}_r(s, m, s_1)$ and $\tilde{f}_m(s, m, s_1)$ of Ex. 5 are safe UBFs for *rpop* and *main* respectively. For example, it is easy to check that

$$\mu_1 = 1, \mu_2 = \mu_5 = \frac{1}{2}, \mu_4 = \mu_6 = \mu_0 = 0, \lambda_1 = 1, \lambda_2 = -1, \lambda_3 = \lambda_4 = \lambda_5 = 0$$

is a model of ξ, which corresponds to the desired UBFs $s \cdot m + \frac{1}{2}(m^2 + m)$ and $(s - s_1) \cdot m$ for procedures *main* and *rpop* respectively. It is worth noting the inequalities $\lambda_1 \geq 1$ and $\lambda_2 = -\lambda_1$, meaning that any UBF for *rpop must* involve both $s \cdot m$ and $s_1 \cdot m$ (recall that s_1 is its output parameter). If we analyse *rpop* alone this would not be the case, and UBFs like $s \cdot m$ would be possible, however, this is essential in order to obtain the quadratic UBF for *main*.

It is important to note that once the constraints over the template parameters (i.e., ξ in the above example) are generated, then one should try to find a model of ξ that results in a tight UBF. This process usually depends on the kind of expression used in the templates. For example, in the case of polynomial templates one could try to first set the parameters of the higher degree components to 0, etc. Another possibility is to start from a polynomial with low degree, and increment it gradually until an UBF is found.

5 Inference of Peak-Cost

When a given program only acquires resources, the net-cost analysis can be used to estimate the amount of resources required for safely executing the program. This, however, is not the case when the program can also release resources. For example, the net-cost of the program in Fig 2 is 0, since all resources are released either at J5 or J7, however, it requires at least $n+1$ resources in order to execute correctly. In order to estimate the amount of resources required for safely executing such programs, what we need is the peak-cost, which is the maximum amount of resources that a program can hold simultaneously.

Definition 5 (safe peak-cost UBFs). *Let p be a procedure with n input parameters. A function $\hat{f}_p : \mathbb{Z}^n \mapsto \mathbb{Q}$ is a safe UBF on the peak-cost of p, iff for any $\langle \bar{v}_1, c\rangle \in \hat{\pi}(p)$ it holds $\hat{f}_p(\bar{v}_1) \geq c$.*

Our approach for inferring UBFs on the peak-cost is done in two steps, verification and inference, similar to the case of net-cost.

Verification of UBFs on the Peak-Cost. Let us start by explaining the basics of the verification step. Assume that we have a procedure p defined by the following single rule

$$p(\bar{x}, \bar{y}) \leftarrow q_1(\bar{x}_1, \bar{y}_1), q_2(\bar{x}_2, \bar{y}_2)$$

and assume that we have UBFs $\hat{f}_{q_1}$ and $\hat{f}_{q_2}$ on the peak-cost of q_1 and q_2 respectively. We are interested in verifying that a given function $\hat{f}_p(\bar{x})$ is indeed a safe UBF on the peak-cost of p. When executing p, the peak-cost might be reached while executing q_1 or q_2. If it is reached during q_1, then the peak-cost of p is like that of q_1, and if it is reached during q_2, then the peak-cost of p is like that of q_2 plus the amount of resources that p holds before calling q_2. Now note that this last amount is exactly the net-cost of q_1. Thus, in order to verify the correctness of $\hat{f}_p$ it is sufficient to check that the condition $\hat{f}_p(\bar{x}) \geq \hat{f}_{q_1}(\bar{x}_1) \wedge \hat{f}_p(\bar{x}) \geq \tilde{f}_{q_1}(\bar{x}_1, \bar{y}_1) + \hat{f}_{q_2}(\bar{x}_1)$ holds for any values of the program variables, where $\tilde{f}_{q_1}(\bar{x}_1, \bar{y}_1)$ is a safe UBF on the *net-cost* of q_1. Applying this principle to all rules of the program, it is possible to verify the correctness of several UBFs simultaneously.

Given a set $\hat{F}$ of candidate UBFs on the peak-cost, which includes a function $\hat{f}_p : \mathbb{Z}^n \mapsto \mathbb{Q}$ for each procedure $p \in P$, we want to build a VC whose validity

implies that each $\hat{f}_p$ is indeed a safe UBF. For this, we assume a given set $\tilde{F}$ of safe UBFs on the net-cost of each procedure (later we will see that $\hat{F}$ and $\tilde{F}$ can be verified or inferred simultaneously). The peak-cost VC, denoted by $\Phi(\tilde{F}, \hat{F})$, is generated from the program rules as we explain next.

Definition 6 (Peak-cost VC). *Let $\hat{F}$ be a set of candidate UBFs on the peak-cost, and $\tilde{F}$ be a set of safe UBFs on the net-cost. For each rule $r \equiv p(\bar{x}, \bar{y}) \leftarrow b_1, b_2, \ldots, b_n$, we generate a condition ϕ_r according to the following steps*

1. *let $b_{\ell_1}, \ldots, b_{\ell_k}$, with $1 \leq \ell_1 < \cdots < \ell_k \leq n$, be all elements of the body that are of the form $q_{\ell_i}(\bar{x}_{\ell_i}, \bar{y}_{\ell_i})$ or $\texttt{acq}(e)$. We assume there is at least one such element, otherwise we add $\texttt{acq}(0)$ at the end of r;*
2. *let φ_i be the conjunction of all constraints in r up to b_{ℓ_i};*
3. *the peak-cost $\hat{b}_{\ell_i}$ of an instruction b_{ℓ_i} is defined as follows: if $b_{\ell_i} \equiv q_{\ell_i}(\bar{x}_{\ell_i}, \bar{y}_{\ell_i})$ then $\hat{b}_{\ell_i} \equiv \hat{f}_q(\bar{x}_{\ell_i})$, and if $b_{\ell_i} \equiv \texttt{acq}(e)$ then $\hat{b}_{\ell_i} \equiv e$;*
4. *let ϕ_r be the formula below where $\bar{w} = vars(r)$ and $\tilde{b}_j$ are as in Def. 4.*

$$\phi_r \equiv \underbrace{(\wedge_{i=1}^{k} \forall \bar{w} : \varphi_i \Rightarrow \hat{f}_p(\bar{x}) \geq (\Sigma_{j=1}^{\ell_i - 1} \tilde{b}_j) + \hat{b}_j))}_{\mathcal{A}} \wedge \underbrace{(\forall \bar{w} : \varphi_1 \Rightarrow \hat{f}_p(\bar{x}) \geq 0))}_{\mathcal{B}}$$

Then, the peak-cost VC is $\Phi(\tilde{F}, \hat{F}) = \wedge_{r \in P} \phi_r$.

Let us explain the parts of ϕ_r: ($\mathcal{A}$) this part generalises the intuition that we have explained before. Intuitively, the instructions $b_{\ell_1}, \ldots, b_{\ell_k}$ are those that might *increase* the resource consumption, thus, the peak-cost of p should be greater than or equal to the peak-cost $\hat{b}_{\ell_i}$ of each b_{ℓ_i} plus the resources $\sum_{j=1}^{\ell_i - 1} \tilde{b}_j$ that p holds before executing $\hat{b}_{\ell_i}$ (note the use of the net-cost $\tilde{b}_j$); and ($\mathcal{B}$) this part requires that the peak function is non-negative. Note that in principle we should require $\forall \bar{w} : \varphi_i \Rightarrow \hat{f}_p(\bar{x}) \geq 0$ for all $i \in [1 \ldots k]$, however, requiring $\mathcal{B}$ is enough since $\varphi_i \Rightarrow \varphi_1$ for all $i \in [2 \ldots k]$. In the examples below we sometimes omit the second part $\mathcal{B}$ when it is redundant.

Example 7. The peak-cost VC for the program of Fig. 2, w.r.t. (some generic) $\tilde{F}$ and $\hat{F}$, is $\Phi(\tilde{F}, \hat{F}) = \phi_{r_1} \wedge \cdots \wedge \phi_{r_5}$ where

$$\begin{aligned}
\phi_{r_1} &\equiv \forall \bar{w}_1 : n = 0 \Rightarrow \hat{f}_p(n) \geq 0 \\
\phi_{r_2} &\equiv (\forall \bar{w}_2 : n \geq 1 \Rightarrow \hat{f}_p(n) \geq \hat{f}_q(n)) \wedge \\
&\quad\ (\forall \bar{w}_2 : n \geq 1 \wedge n_1 = n - 1 \Rightarrow \hat{f}_p(n) \geq \tilde{f}_q(n, m) - m + \hat{f}_p(n_1)) \wedge \\
&\quad\ (\forall \bar{w}_2 : n \geq 1 \Rightarrow \hat{f}_p(n) \geq 0) \\
\phi_{r_3} &\equiv \forall \bar{w}_3 : i \geq 0 \Rightarrow \hat{f}_l(i) \geq 2 \\
\phi_{r_4} &\equiv (\forall \bar{w}_4 : i \geq 1 \Rightarrow \hat{f}_l(i) \geq 2) \wedge (\forall \bar{w}_4 : i \geq 1 \wedge i_2 = i - 1 \Rightarrow \hat{f}_l(i) \geq 2 + \hat{f}_l(i_2)) \\
\phi_{r_5} &\equiv (\forall \bar{w}_5 : n \geq 1 \wedge i = \tfrac{n}{2} \Rightarrow \hat{f}_q(n) \geq \hat{f}_l(i)) \wedge (\forall \bar{w}_5 : n \geq 1 \wedge i = \tfrac{n}{2} \Rightarrow \hat{f}_q(n) \geq 0)
\end{aligned}$$

Formula ϕ_{r_2}, for example, corresponds to the second rule of procedure p. It consists of 3 subformulas, the first two are the $\mathcal{A}$-part and the last is the $\mathcal{B}$-part. In the second, note the expression $\tilde{f}_q(n, m) - m$ which is the amount of resource that p holds before the recursive call to p. Using $\tilde{f}_q(n, m) = 2{\cdot}m$, $\hat{f}_p(n) = n + 2$,

$\hat{f}_q(n) = n+2$, and $\hat{f}_l(i, i_1) = 2{\cdot}i_1 + 2$, it is possible to verify that $\Phi(\tilde{F}, \hat{F})$ is valid. However, using another safe UBF on the net-cost of q, e.g., $\tilde{f}_q(n, m) = n + 2$, then $\Phi(\tilde{F}, \hat{F})$ is not valid. Indeed, $2{\cdot}m$ is the most precise UBF on the net-cost of q, and is the one needed to verify the above UBF on the peak-cost of p.

Theorem 2. *Given a set $\tilde{F}$ of safe UBFs on the net-cost (Th. 1), and a set $\hat{F}$ of candidate UBFs on the peak-cost, if $\models \Phi(\tilde{F}, \hat{F})$, then $\hat{F}$ is safe.*

As in the case of $\Psi(\tilde{F})$ cost, checking the validity of $\Phi(\tilde{F}, \hat{F})$ reduces to a satisfiability problem of first order logic.

Inferring UBFs on the Peak-Cost. Our main interest is in inferring $\hat{F}$ rather than verifying the correctness of a given one. This can be done using template UBFs as the case of net-cost. However, an important point is that instead of assuming a given set $\tilde{F}$ of UBFs on the net-cost, we can infer it at the same time as $\hat{F}$, simply by considering the VC $\Phi(\tilde{F}, \hat{F}) \wedge \Psi(\tilde{F})$. This is actually essential in practice, since as we have seen in Ex. 7 not any safe UBF on the net-cost can be used to infer the peak-cost. Inferring them simultaneously will force choosing the required one.

Example 8. Let $\tilde{F}$ and $\hat{F}$ be defined by the following linear UBF templates:

$$\begin{array}{lll} \tilde{f}_p(n) = \lambda_1{\cdot}n + \lambda_2 & \tilde{f}_q(n,m) = \lambda_3{\cdot}n + \lambda_4{\cdot}m + \lambda_5 & \tilde{f}_l(i,i_1) = \lambda_6{\cdot}i + \lambda_7{\cdot}i_1 + \lambda_8 \\ \hat{f}_p(n) = \mu_1{\cdot}n + \mu_2 & \hat{f}_q(n) = \mu_3{\cdot}n + \mu_4 & \hat{f}_l(i) = \mu_5{\cdot}i + \mu_6 \end{array}$$

and let $\Phi(\tilde{F}, \hat{F})$ be the VC of Ex. 7 using these $\tilde{F}$ and $\hat{F}$. Moreover, let $\Psi(\tilde{F}) = \psi_{r_1} \wedge \cdots \wedge \psi_{r_5}$ be the corresponding net-cost VC using the above $\tilde{F}$, where

$$\begin{array}{l} \psi_{r_1} \equiv \forall \bar{w}_1 : n = 0 \Rightarrow \tilde{f}_p(n) \geq 0 \\ \psi_{r_2} \equiv \forall \bar{w}_2 : n \geq 1 \wedge n_1 = n - 1 \Rightarrow \tilde{f}_p(n) \geq \tilde{f}_q(n, m) - m + \tilde{f}_p(n_1) + m \\ \psi_{r_3} \equiv \forall \bar{w}_3 : i \geq 0 \wedge i_1 = i - 1 \Rightarrow \tilde{f}_l(i, i_1) \geq 2 \\ \psi_{r_4} \equiv \forall \bar{w}_4 : i \geq 1 \wedge i_2 = i - 1 \Rightarrow \tilde{f}_l(i, i_1) \geq 2 + \tilde{f}_l(i_2, i_1) \\ \psi_{r_5} \equiv \forall \bar{w}_5 : n \geq 1 \wedge i = \frac{n}{2}, m = i - i_1 \Rightarrow \tilde{f}_q(n, m) \geq \tilde{f}_l(i, i_1) \end{array}$$

Then, applying quantifier elimination on $\Phi(\tilde{F}, \hat{F}) \wedge \Psi(\tilde{F})$ to eliminate the universally quantified variables, we get a formula ξ over the template parameters that is a conjunction of the following equalities and inequalities

$$\begin{array}{l|l|l|l|l} \lambda_2 \geq 0 & \lambda_3 = 0 & \lambda_8 \leq \lambda_5 \leq 0 & \lambda_1 = \lambda_4 - 2 & \lambda_6 + \lambda_8 \geq 2 \\ \lambda_6 \geq 2 & \mu_6 \geq 2 & \mu_1 = \lambda_1 + 1 & \lambda_4 = \lambda_6 = -\lambda_7 & 2{\cdot}\mu_6 + \mu_5 \leq 2{\cdot}\mu_4 + 2{\cdot}\mu_3 \\ \mu_5 \geq 2 & \mu_2 \geq 0 & 2{\cdot}\mu_3 \geq \mu_5 & \lambda_6 \geq \mu_3 + 1 & \mu_1 + \mu_2 \geq \mu_3 + \mu_4 \end{array}$$

The models of ξ define possible instantiations $\tilde{F}$ and $\hat{F}$ such that they are safe UBFs. E.g., there is a model of ξ with $\mu_1 = 1$ and $\mu_2 = 2$ which defines the UBF $n + 2$ on the peak-cost of p. Note the constraint $\lambda_3 = 0$, which means that the UBF on the net-cost of q must not depend on the input n (in Ex. 7 we failed with $\tilde{f}_q(n, m) = n + 2$). This demonstrates how the peak-cost VC affects the net-cost one. Note that, for p, we have inferred the UBF $n + 2$ and not the optimal one $n + 1$ because the quantifier elimination is done over $\mathbb{R}$ and not over $\mathbb{Z}$.

Example 9. Let us finish with an example of a non-terminating program. Consider the following (contrived) program, which is defined by a single rule

$$p([n],[y_1]) \leftarrow n \geq m \geq 0, \mathtt{acq}(m), n_1 = n - m, p([n_1],[y_1]).$$

Procedure p receives a non-negative integer n, non-deterministically chooses a non-negative value $m \leq n$, acquires m resources, and then calls p recursively with $n - m$. The peak-cost of this program is exactly n, since any infinite trace cannot acquire more than n resources and there are infinite traces that acquire exactly n. The peak-cost VC for this program is

$$(\forall n, m : n \geq m \geq 0 \Rightarrow \hat{f}_p(n) \geq m) \wedge (\forall n, m : n \geq m \geq 0 \Rightarrow \hat{f}_p(n) \geq m + \hat{f}_p(n_1)$$

Assuming the template UBF $\hat{f}_p(n) = \lambda_1 \cdot n + \lambda_2$, the elimination of the universally quantified variables result in the formula $\xi = \lambda_1 \geq 1 \wedge \lambda_2 \geq 0$. Since $\lambda_1 = 1$ and $\lambda_2 = 0$ is a model of ξ, then $\hat{f}_p(n) = n$ is a safe UBF.

6 Relation to Amortised Cost Analysis

In this section we discuss an interesting relation that we have observed between UBFs that are defined in terms of both input and output parameters, and the notion of potential functions used in the context of amortised cost analysis. This may provide a semantics-based explanation to why amortised analysis can obtain more precise UBFs.

A potential function, in the context of an ACR, is a function that maps a given state to a non-negative rational number, which is called the potential of the state. This potential can be interpreted as the amount of resources available in the given state. An automatic amortised cost analysis [15] assigns to each procedure $p(\bar{x}, \bar{y})$ two potential functions: *input* $P_p(\bar{x})$, and *output* $Q_p(\bar{y})$. Intuitively, the input potential $P_p(\bar{x})$ must be large enough to pay for the cost of executing $p(\bar{x}, \bar{y})$, and, upon exit, leaving at least $Q_p(\bar{y})$ resources to be consumed later. Thus, if c is the net-cost of p, then $P_p(\bar{x}) \geq c + Q_p(\bar{y})$ must hold. This later expression can be rewritten as $P_p(\bar{x}) - Q_p(\bar{y}) \geq c$, which means that $P_p(\bar{x}) - Q_p(\bar{y})$ is an UBF on the net-cost of p, but also is an UBF that uses input and the output parameters. Thus, the above potential functions are in principle UBFs as defined in Def. 3, however, they are just a special case since $P_p(\bar{x}) - Q_p(\bar{y})$ does not allow using, for example, expressions like $s_1 \cdot m$.

We have tried to analyse (a functional version of) the Stack example using the amortised analysis of [15], which uses the above notion of potential functions. The analysis failed to obtain the expected quadratic UBF, and instead, it reported a cubic UBF. This failure confirms that it is essential to define the output potential for *rpop* as $s_1 \cdot m$, which cannot be defined using the above kind of potential functions. Note that this should not be interpreted as a fundamental limitation of [15], since their underlying machinery can be easily adapted to support potential functions of this form. In addition, the above discussion should be considered only in the context of the ACR language, since amortised analysis has many other features that goes beyond the ACR language.

7 Implementation and Experiments

A prototype implementation of our analysis is available at `http://costa.ls.fi.upm.es/acrp`. It receives as input an ACR program and a set of template UBFs. Then, it generates the VCs described in Secs. 4 and 5 as a REDUCE script [20], executes the script to eliminate the universally quantified variables, and finally outputs the template parameters constraints in SMT2-LIB format, which can be then solved using off-the-shelf SMT solvers.

For the quantifier elimination, the REDUCE script uses the REDLOG package [11] with the theory of *real closed fields*. This theory allows using a wide range of template UBFs, such as multivariate polynomial, max and min operations, etc. As done in [19], REDLOG can be switched to use SLFQ [7], which is a formula simplifier for the theory of real closed fields. Using SLFQ significantly reduces the size of the template parameters constraints, and thus improves the overall performance. For solving the template parameters constraints we have used Z3 [21], employing the logic of *non linear real arithmetic* (QF_NRA). Currently, we only ask the SMT solver for a satisfying assignment, which in turn instantiate the templates to safe UBFs. Looking for an assignment that gives the tightest UBFs is left for future work.

We have applied the analyser on small examples collected from cost analysis literature. All are available in the above address. For these examples we obtained the expected precise UBFs. Unfortunately, being based on real quantifier elimination, our procedure does not yet scale for large programs. In a future work we plan to explore patterns of ACR programs for which (a variation of) the analysis scales, e.g., for the case of the multivariate polynomials of [15].

8 Related Work

Static cost analysis dates back to the seminal work of Wegbreit [22]. Recently it has received a considerable attention which resulted in several cost analysers for different programming languages [2,10,12,15]. The research in this paper is mostly related to [2] and [15], in the sense that our research was motivated by the limitations of [2], and our solution turned to have common ideas with of [15] as we have explained in Sec. 6. When comparing [15], the advantage of our analysis is in that it has a more general notion of potential functions, it is not limited to polynomial templates, and can handle variables with negative values. However, unlike ours, their techniques can handle data-structures by assigning potentials to its parts, and their tool is reasonably scalable and performs very well in practice.

Our peak-cost constraints are similar to those of [3], they were used for inferring memory consumption in the presence of garbage collection. The limitations of CRs have been considered also in [4], but from a different perspective. Solving CRs using template function and real quantifier elimination has been considered before in [5]. However, it cannot handle the limitations we pointed out in this paper, and cannot handle non-terminating programs. Also [13,23] deal with

similar problems, however, they cannot handle the limitation described in this paper, and cannot handle non-terminating programs. Real quantifier elimination has been used for program verification in [8,16,18].

9 Conclusions

In this paper we have studied well known limitations of cost analysis approaches that are based on the use of CRs. We have shown that, unlike it was assumed so far, the reason for these limitations is that CRs ignore the output values of procedures. In particular, we have shown that there are programs whose cost cannot be modeled precisely using CRs. In order to overcome these limitations, we have defined the notion of UBFs that use both input and output parameters, and developed a novel approach for cost analysis that is based on this kind of UBFs. Interestingly, we have found a relation between this kind of UBFs and potential functions that are used in automatic amortised cost analysis [15], which might give an alternative explanation to why amortised analysis (of ACR programs) can be more precise than the classical approach.

Starting from template UBFs, our analysis generates a verification condition over these templates in which the program variables are universally quantified. Eliminating these variables using quantifier elimination tools results in a (possibly non-linear constraint) whose models define possible instantiations for the templates such that they are safe UBFs. An important feature of approach is that it can be used for inferring lower-bounds (for terminating programs) with minimal changes: just replacing $\geq$ by $\leq$ in the VC, and, in addition, $\wedge$ by $\vee$ in each peak-cost condition ϕ_r. Due to lack of space we skipped the details. We have also reported on a preliminary implementation and its evaluation on small examples. For future work, we would like to find some special cases of ACR program for which the analysis can scale to large programs.

Acknowledgements. This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 and PRI-AIBDE-2011-0900 projects, by UCM-BSCH-GR35/10-A-910502 grant and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project. Diego Alonso is supported by the UCM PhD scholarship program.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-Form Upper Bounds in Static Cost Analysis. Journal of Automated Reasoning 46(2), 161–203 (2011)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Object-Oriented Bytecode Programs. Theoretical Computer Science 413(1), 142–159 (2012)

3. Albert, E., Genaim, S., Gómez-Zamalloa, M.: Parametric Inference of Memory Requirements for Garbage Collected Languages. In: ISMM, pp. 121–130. ACM, New York (2010)
4. Albert, E., Genaim, S., Masud, A.N.: More Precise Yet Widely Applicable Cost Analysis. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 38–53. Springer, Heidelberg (2011)
5. Anderson, H., Khoo, S.-C., Andrei, Ş., Luca, B.: Calculating Polynomial Runtime Properties. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 230–246. Springer, Heidelberg (2005)
6. Benoy, F., King, A.: Inferring Argument Size Relationships with CLP(R). In: Gallagher, J.P. (ed.) LOPSTR 1996. LNCS, vol. 1207, pp. 204–223. Springer, Heidelberg (1997)
7. Brown, C.W., Gross, C.: Efficient Preprocessing Methods for Quantifier Elimination. In: Ganzha, V.G., Mayr, E.W., Vorozhtsov, E.V. (eds.) CASC 2006. LNCS, vol. 4194, pp. 89–100. Springer, Heidelberg (2006)
8. Chen, Y., Xia, B., Yang, L., Zhan, N., Zhou, C.: Discovering Non-linear Ranking Functions by Solving Semi-algebraic Systems. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 34–49. Springer, Heidelberg (2007)
9. Cormen, T.H., Leiserson, C.E., Rivest, R., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press (2009)
10. Debray, S.K., Lin, N.-W.: Cost Analysis of Logic Programs. ACM Transactions on Programming Languages and Systems 15(5), 826–875 (1993)
11. Dolzmann, A., Sturm, T.: REDLOG: Computer Algebra meets Computer Logic. ACM SIGSAM Bulletin 31(2), 2–9 (1997)
12. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In: Proc. of POPL 2009, pp. 127–139. ACM (2009)
13. Gulwani, S., Zuleger, F.: The Reachability-Bound Problem. In: PLDI, pp. 292–304. ACM (2010)
14. Hickey, T.J., Cohen, J.: Automating Program Analysis. J. ACM 35(1), 185–220 (1988)
15. Hofmann, M., Hoffmann, J., Aehlig, K.: Multivariate Amortized Resource Analysis. In: POPL 2011, pp. 357–370. ACM (2011)
16. Kapur, D.: Automatically generating loop invariants using quantifier elimination. In: Deduction and Applications, vol. 05431 (2006)
17. Le Métayer, D.: ACE: An Automatic Complexity Evaluator. ACM Trans. Program. Lang. Syst. 10(2), 248–266 (1988)
18. Monniaux, D.: Automatic modular abstractions for template numerical constraints. Logical Methods in Computer Science 6(3) (2010)
19. Sturm, T., Tiwari, A.: Verification and Synthesis using Real Quantifier Elimination. In: ISSAC 2011, pp. 329–336. ACM (2011)
20. REDUCE Computer Algebra System. REDUCE home page
21. Z3 Theorem Prover. Z3 home page
22. Wegbreit, B.: Mechanical Program Analysis. Communications of the ACM 18(9) (1975)
23. Zuleger, F., Gulwani, S., Sinn, M., Veith, H.: Bound Analysis of Imperative Programs with the Size-Change Abstraction. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 280–297. Springer, Heidelberg (2011)

A Proofs

In this section we provide the proofs for Theorems 1 and 2.

1. Since $\tilde{\tau}(t)$ is a function over the set of *terminated complete traces*, we define for this set a well-founded order, based on the notion of call subtrace nesting, and prove Theorem 1 by induction on this WFO.
2. Since $\hat{\tau}(t)$ is defined over the set of *prefixes of complete traces*, we define another WFO for this set and prove Theorem 2 by induction on this WFO.

We first introduce some (fairly standard) notations and properties. For a finite trace t, $|t|$ denotes the length of t. The concatenation (a.k.a. junction) of a finite trace $t_1 \equiv s_1 \dots s_n$ and a trace $t_2 \equiv s_n s_{n+1} \dots$ is the trace $t_1 \cdot t_2 \equiv s_1 \dots s_n s_{n+1} \dots$, that performs the transitions of t_2 after those of t_1. t_1 is a subtrace of t_2 (written $t_1 \sqsubseteq t_2$) if there is a finite trace t_p and a trace t_s such that $t_2 = t_p \cdot t_1 \cdot t_s$. If t_p (resp. t_s) is the empty trace, then t_1 is a prefix (resp. suffix) of t_2. The relation $\sqsubseteq$ defines a well founded order over the set of finite traces. Clearly, if $t_1 \sqsubseteq t_2$ and $t_1 \neq t_2$, then $|t_1| < |t_2|$.

We are interested in traces that represent the evaluation of calls to procedures $p(\bar{x}, \bar{y})$ and $q(\bar{w}, \bar{z})$ of the ACR program P. In addition to the notions of execution, terminated execution and complete execution from Section 2, we consider subtraces that correspond to internal executions. Given a trace t, a *subexecution* of q in t is a subtrace t_q of t that starts at a an intermediate state $\langle \varphi, q(\bar{x}, \bar{y}) \cdot \bar{b} \rangle$ of t, and it is a *complete subexecution* if it ends at an intermediate state $\langle \psi, \cdot \bar{b} \rangle$ of t, where $\bar{b}$ is any sequence of pending instructions. Intuitively, it is the subtrace that deals with evaluating a call.

Recall that the verification conditions for each rule are built by replacing each call to $q(\bar{x}, \bar{y})$ with either $\tilde{f}_q(\bar{x}, \bar{y})$ or $\hat{f}_q(\bar{x})$. This is correct if **(1)** for any execution t of p, for any subexecution t_q of q in t, the net cost (resp. peak cost) is contained in $\tilde{\pi}(q)$ (resp. $\hat{\pi}(q)$); and **(2)** the verification condition correctly combines the cost of each subexecution. Each point is expressed in one of these properties:

Property 1. Let $\bar{b}_1, \bar{b}_2$ be sequences of instructions and φ, ϕ, ψ constraints.

- For any sequence $\bar{b}_2$, it holds that $\langle \varphi, \bar{b}_1 \cdot \bar{b}_2 \rangle \overset{*}{\rightsquigarrow} \langle \psi, \bar{b}_2 \rangle$ if and only if $\langle \varphi, \bar{b}_1 \rangle \overset{*}{\rightsquigarrow} \langle \psi, \epsilon \rangle$.
- If ϕ is a constraint such that $vars(\phi) \cap vars(\bar{b}_1, \varphi) = \emptyset$, and $\varphi \wedge \phi$ is equivalent to φ, then it holds that $\langle \varphi \wedge \phi, \bar{b}_1 \rangle \overset{*}{\rightsquigarrow} \langle \psi \wedge \phi, \epsilon \rangle$ if and only if $\langle \varphi, \bar{b}_1 \rangle \overset{*}{\rightsquigarrow} \langle \psi, \epsilon \rangle$.

So, any subexecution $t_q \sqsubseteq t$ of t corresponds to an execution of q, and therefore $\tilde{\tau}(t_q) \in \tilde{\pi}(q)$ and $\hat{\tau}(t_q) \in \hat{\pi}(q)$. Next property allows us to obtain the cost of a trace concatenation as a function the costs of each trace.

Property 2. For the finite traces t_1, t_2 and the infinite trace t_3, it holds that $\tilde{\tau}(t_1 \cdot t_2) = \tilde{\tau}(t_1) + \tilde{\tau}(t_2)$ and that $\hat{\tau}(t_1 \cdot t_3) = max\,(\hat{\tau}(t_1), \hat{\tau}(t_3) + \tilde{\tau}(t_1))$.

This property is also extended to the cost of procedures.

Property 3. Let $p(\bar{x}, \bar{y})$ be a procedure defined with the single rule $p(\bar{x}, \bar{y}) \leftarrow q_1(\bar{x}_1, \bar{y}_1), q_2(\bar{x}_2, \bar{y}_2)$. then it holds that

- For every $\langle \bar{v}_1, \bar{v}_2, d\rangle \in \tilde{\pi}(p)$, there exists a $\langle \bar{v}_1, \bar{v_3}, d_1\rangle \in \tilde{\pi}(q_1)$ and a $\langle \bar{v}_3, \bar{v_2}, d_2\rangle \in \tilde{\pi}(q_2)$ such that $d = d_1 + d_2$.
- For every $\langle \bar{v}_1, d\rangle \in \hat{\pi}(p)$, either **(a)** exists $\langle \bar{v}_1, d_1\rangle \in \hat{\pi}(q_1)$ such that $d = d_1$; or **(b)** exists $\langle \bar{v}_1, \bar{v}_2, d_1\rangle \in \tilde{\pi}(q)_1$ and $\langle \bar{v}_2, d_2\rangle \in \hat{\pi}(q_2)$ such that $d = d_1 + d_2$.

We write $\tilde{\mathbb{P}}$ to denote the set of complete terminated executions of the predicates of the ACR P. Note that $\tilde{\mathbb{P}}$ is a subset of the set of finite traces, and therefore $\sqsubseteq$ is a well founded order on $\tilde{\mathbb{P}}$. Let $\tilde{r}_{\sqsubseteq}$ be a function in $\tilde{\mathbb{P}} \mapsto \mathbb{N}$ that, for any trace t gives the length of the longest chain $\tau_\epsilon \sqsubseteq t_1 \sqsubseteq \cdots \sqsubseteq t$ of direct subexecution relations, i.e. the depth of the deepest nested call.

Proof (Proof of Theorem 1). Theorem 1 states that, for each procedure $p(\bar{x}, \bar{y})$, for each $\langle \bar{v}_1, \bar{v}_2, d\rangle \in \tilde{\pi}(p)$, it holds that $d \leq \tilde{f}_p(\bar{v}_1 \bar{v}_2)$. Unfolding Def. 1, that means that for every $t \in \tilde{\mathbb{P}}$, such that $t \equiv \langle \bar{x} = \bar{v}_1, p(\bar{x}, \bar{y})\rangle \overset{*}{\rightsquigarrow} \langle \psi \wedge \bar{y} = \bar{v}_2, \epsilon\rangle$, it holds that $\tilde{\tau}(t) \leq \tilde{f}_p(\bar{v}_1, \bar{v}_2)$. Since $\Psi(\tilde{F})$ is obtained in a rule-wise manner, we assume a generic rule $p(\bar{x}, \bar{y}) \leftarrow b_1, b_2, \ldots, b_n$, and refine the proof for all terminated traces of p that start by unfolding the call to p with that rule. We define $\bar{w}$ as the set of all variables in the rule, exp as the sum of all the expressions in the `acq`() instructions minus the sum of those inside `rel`() instructions, and φ as the conjunction of all constraints instructions in the rule. Note that the constraint at the final state of t, must hold that $\psi \models \varphi$. Therefore, if $\bar{w}$ satisfy φ, then there is a valid execution of p that ends with those values.

Since $\tilde{r}_{\sqsubseteq}$ is a total function in $\tilde{\mathbb{P}}$, we can prove the property for every terminated trace t by induction on the value of $\tilde{r}_{\sqsubseteq}$.

Base case $\tilde{r}_{\sqsubseteq}(t) = 0$ In the base case, t corresponds to a rule such that no b_i is a call instruction. The value of $\tilde{\tau}(t)$ is the value of exp for some assignment of $\bar{w}$ that satisfies φ. Therefore, if for all assignments σ to the variables $\bar{w}$ such that $\sigma \models \varphi$, it holds that $\tilde{f}_p \geq exp$, then for every execution t of p with that rule, at the final state of t it holds that $\psi \models \tilde{f}_p \geq \tilde{\tau}(t)$.

Inductive case $\tilde{r}_{\sqsubseteq}(t) > 0$ In this case, t starts unfolding the call to $p(\bar{x}, \bar{y})$ with a rule such that contains k call instructions to predicates q_i. For each q, trace t must have a subexecution $t_q \sqsubseteq t$ of q, such that $\tilde{r}_{\sqsubseteq}(t_q) < \tilde{r}_{\sqsubseteq}(t)$. Applying Prop. 2, we can see that the cost of t must be equal to exp and $\sum_{i=1}^{k} \tilde{\tau}(t_{q_i})$. We assume, as inductive hypothesis, that for each predicates q_i, for all execution traces t_q of q such that $\tilde{r}_{\sqsubseteq}(t_q) < k$, it holds that $\tilde{f}_q \geq \tilde{\tau}(t)$. Since the sum is monotonic, $\sum_{i=1}^{k} \tilde{\tau}(t_{q_i}) \leq \sum_{i=1}^{k} \tilde{f}_{q_i}$. As before, if t ends at $\langle \psi, \epsilon\rangle$, it must hold that $\psi \models \varphi$. So, if $\varphi \models \tilde{f}_p(\bar{x}, \bar{y}) \geq exp + \sum_{i=1}^{k} \tilde{f}_{q_i}$, then for every trace t, $\tilde{f}_p \geq \tilde{\tau}(t)$. □

The proof for Theorem 2 proceeds in a similar manner, only that instead of $\tilde{\mathbb{P}}$, we perform induction over the set $\hat{\mathbb{P}}$ of finite prefixes of complete executions of predicates of P. Note that, although it also includes prefixes of *infinite* traces,

$\hat{\mathbb{P}}$ is a subset of the set of finite traces. Therefore, $\sqsubseteq$ is a well founded order also in $\hat{\mathbb{P}}$, so this proof can also proceed by induction. Just as we defined $\tilde{r}_{\sqsubseteq}$ as a function in $\tilde{\mathbb{P}} \mapsto \mathbb{N}$, so we define now $\hat{r}_{\sqsubseteq} : \hat{\mathbb{P}} \mapsto \mathbb{N}$, that for each finite prefix t gives the length of the longest subexecution nesting chain.

Two points in the construction of $\Phi(\tilde{F}, \hat{F})$ require a special explanation. The first one is constructing, for each rule, a conjunction of verification conditions, for each instruction that may reach the peak consumption (calls or `acq`()). The actual verification condition is $\hat{f}_p(\bar{x}) \geq max_{i=1}^{n} \hat{b}_i + \sum_{j=0}^{i-1} \tilde{b}_j$, which is equivalent to $\wedge_{i=1}^{n} \hat{f}_p \geq \hat{b}_i + \sum_{j=0}^{i-1} \tilde{b}_j$. The second one is ignoring prefixes that end before a constraint or a `rel`() instruction, which can be done due to this property.

Property 4. If $t_1 \in \hat{\mathbb{P}}$ is a finite prefix $t_1 \equiv s \overset{*}{\rightsquigarrow} \langle \psi, \bar{b}_1 \cdot \bar{b}_2 \rangle$ of a complete trace and $\bar{b}_1$ only contains constraints and release codes, then there is a trace $t_2 \in \hat{\mathbb{P}}$ such that $t_2 \equiv s \overset{*}{\rightsquigarrow} \langle \psi', \bar{b}_2 \rangle$ and $\hat{\tau}(t_2) = \hat{\tau}(t_1)$.

Intuitively, if t_1 is a prefix of a complete trace t, then that complete trace must execute all the codes in b_2. Note that in Def. 1, $\hat{\pi}(p)$ only considers finite prefixes of complete traces, instead of all finite traces from $\langle \bar{x} = \bar{v}, p(\bar{x}, \bar{y}) \rangle$.

Proof (Proof of Theorem 2). The proof proceeds almost the same as the proof for Theorem 2. The major difference is in the inductive step: when t is a prefix of a complete trace of p such that $\hat{r}_{\sqsubseteq}$ the trace instances a rule with $k > 0$ calls. It must not assume that for each call there is a terminated subexecution t_q. Instead, it must assume that

- t has executed all instructions up to any one of the calls q_i; and
- for q_i there is a finite trace $t_i \in \hat{\mathbb{P}}$ that is a suffix of t and a prefix of a complete subexecution of q_i. This t_i holds that $\hat{r}_{\sqsubseteq}(t_i) < \hat{r}_{\sqsubseteq}(t)$.
- For each q_j with $1 \leq j \leq i-1$, there is a $t_j \in \tilde{\mathbb{P}}$ that is a complete subexecution of q_j in t, and $\tilde{r}_{\sqsubseteq}(t_j) < \hat{r}_{\sqsubseteq}(t)$.

□

Precise Cost Analysis via Local Reasoning

Diego Esteban Alonso-Blas, Puri Arenas, and Samir Genaim

DSIC, Complutense University of Madrid (UCM), Spain

Abstract. The classical approach to static cost analysis is based on first transforming a given program into a set of cost relations, and then solving them into closed-form upper-bounds. The quality of the upper-bounds and the scalability of such cost analysis highly depend on the precision and efficiency of the solving phase. Several techniques for solving cost relations exist, some are efficient but not precise enough, and some are very precise but do not scale to large cost relations. In this paper we explore the gap between these techniques, seeking for ones that are both precise and efficient. In particular, we propose a novel technique that first splits the cost relation into several *atomic* ones, and then uses precise local reasoning for some and less precise but efficient reasoning for others. For the precise local reasoning, we propose several methods that define the cost as a solution of a universally quantified formula. Preliminary experiments demonstrate the effectiveness of our approach.

1 Introduction

Static Cost analysis (a.k.a. resource usage analysis) aims at *statically* determining the amount of resources (e.g., memory, execution steps, etc.) required to execute a given program safely, i.e., without running out of resources. Applications of cost analysis range from detecting performance bottlenecks at the development stage, to providing resource consumption guarantees at runtime.

Several cost analysis frameworks exist [4,13,15,10]. Although different in their underlying techniques, they all report the cost as a closed-form upper-bound function (*UB* for short) in terms of the input parameters. This paper uses the classical approach of Wegbreit [18], in particular its extension for Java bytecode [4], where the analysis is carried out in two phases: (1) the input program is transformed into a set of *cost relations* (*CRs* for short) that define its cost; and (2) the *CRs* are solved into *UBs*. While the first phase depends on the programming language in which the program is written [4,11,12,9,17], the second phase is common to all analyses that are based on this approach. In this paper *we focus on the second phase*, i.e., on developing techniques for solving *CRs*. However, we provide enough details to clarify how *CRs* are related to programs.

Example 1. The Java class depicted in Fig. 1 implements a dynamic array, where field data is used to store its elements, and field size represents the number of such elements. Method add adds the elements of the array elems to the dynamic array. When the array data is full (L6), it is replaced by a new one of double

D. Van Hung and M. Ogawa (Eds.): ATVA 2013, LNCS 8172, pp. 319–333, 2013.

320 D.E. Alonso-Blas, P. Arenas, and S. Genaim

```
1 class DynamicArray {
2   int[ ] data;
3   int size;
4   void add(int[ ] elems) {
5     for (int i=0; i<elems.length; i++) {
6       if (data.length == size) {
7         int[ ] tmp = new int[2*data.length];
8         copy(tmp,size,data);
9         data = tmp;
10      }
11      data[size] = elems[i];
12      size++;
13    }
14  }
15  int r;
16  void qsort() {
17    qs(0, size-1);
18  }
19  void qs(int from, int to) {
20    if (to - from < r)
21      insertionSort(from,to);
22    else {
23      int m=partition(from,to);
24      qs(from,m-1);
25      qs(m+1,to);
26    }
27  }
28 }
```

$add(e,d,s) = for(e,d,s,i)$	φ_0=$\{e{\geq}0, d{\geq}0, s{\geq}0, i{=}0\}$
$for(e,d,s,i) = 0$	φ_1=$\{i{\geq}e\}$
$for(e,d,s,i) = 2{\cdot}\mathsf{nat}(s) + 2 + for(e,d',s',i')$	φ_2=$\{i{<}e, s{=}d, d'{=}2{\cdot}d, s'{=}s{+}1, i'{=}i{+}1\}$
$for(e,d,s,i) = 2 + for(e,d,s',i')$	φ_3=$\{i{<}e, d{>}s, s'{=}s{+}1, i'{=}i{+}1\}$
$qsort(s,r) = qs(f,t,r)$	$\psi_0 = \{s{\geq}0, f{=}0, t{=}s{-}1\}$
$qs(f,t,r) = \mathsf{nat}(t{-}f)^2$	$\psi_1 = \{t{-}f{<}r, r{\geq}0\}$
$qs(f,t,r) = \mathsf{nat}(t{-}f) + qs(f,m',r) + qs(m'',t,r)$	$\psi_2 = \{t{-}f{\geq}r, r{\geq}0, f{\leq}m{\leq}t, m'{=}m{-}1, m''{=}m{+}1\}$

Fig. 1. Above, JAVA code of a DynamicArray class. Below, the *CRs* of the methods

size (L7-9). Methods qsort and qs sort the array using a variation of *Quick Sort*, which resorts to *Insertion Sort* when the segment to be sorted is shorter than a threshold defined by field r. Methods copy, partition, and insertionSort are omitted.

Below the JAVA code we show the corresponding *CRs*, generated using a cost model that counts array accesses. Let us explain the *CR* of method add. Variables e, d, s, and i stand for the lengths of arrays elems and data and the values of size and i. Expression $\mathsf{nat}(e)$ is an abbreviation for $\max\{e, 0\}$. The first equation states that the cost of $add(e,d,s)$ is as that of $for(e,d,s,i)$. The constraints on the right impose conditions and relations on the variables. The second equation is for the case of exiting the loop ($i \geq e$). The third one is for the case in which the array is resized. In such case the cost is $2{\cdot}\mathsf{nat}(s)$ (the cost assumed for copy), plus 2 (the accesses at L11), plus the cost of the remaining iterations $for(e,d',s',i')$. Note that $d'{=}2{\cdot}d$ states that the size of array data is doubled. The fourth equation describes the case in which the array is not resized. The equations of qsort are defined similarly. We note that $\mathsf{nat}(t{-}f)^2$ and $\mathsf{nat}(t{-}f)$ correspond to the cost of insertionSort and partition respectively. The constraint $f{\leq}m{\leq}t$ in ψ_2 is an input-output summary inferred for the value m returned by method partition. Methods add and qsort, respectively, have linear and quadratic worst-case complexity. □

Early works on cost analysis [11,9] relied on Computer Algebra Systems (CAS) for solving *CRs*. They can only handle cases in which the *CRs* can be trans-

formed into *recurrence equations* (the only valid input for CAS). This, however, is a very limited subset because *CRs* allow using constraints to define complex applicability conditions and relations between the variables. To overcome this limitation, recent works [3,6] have developed dedicated tools for solving *CRs* into *UBs*. They are mostly based on the use of program analysis techniques. These works are our starting point.

The techniques of [3] are based on assuming worst-case behaviour for all loop iterations. It is very efficient and can handle a wide class of *CRs*. To solve the *CR for*, this technique infers that $2{+}2{\cdot}\mathsf{nat}(e{+}s{-}1)$ is an *UB* on the cost of any iteration of *for*, and it infers that there is at most $\mathsf{nat}(e{-}i)$ iterations of *for*, from which it concludes that $\mathsf{nat}(e{-}i){\cdot}(2{+}2{\cdot}\mathsf{nat}(e{+}s{-}1))$ is an *UB* for *CR for*. Note that this is a quadratic *UB* while the actual cost is linear. In the case of qsort, the loss of precision is even bigger. It first infers that $\mathsf{nat}(t{-}f)^2$ is an *UB* on the cost of each call to qs, and that there are at most $2^{\mathsf{nat}(t-f)}$ of such calls. Then, it concludes that $(t{-}f)^2{\cdot}2^{\mathsf{nat}(t-f)}$ is an *UB* for *CR qs*, while the actual cost is quadratic.

The above imprecision issue, among others, was addressed in [6] where precise and novel techniques for solving *CRs* were proposed. They are based on defining the cost as a solution of a corresponding first-order universally quantified formula. This method, as expected, would obtain the most precise *UBs* for the *CRs for* and *qs*, however, it has two major limitations: (1) a template *UB* has to be provided by the user; and (2) the use of a quantifier elimination procedure for real numbers renders the technique impractical.

In this paper we explore the gap between [3] and [6], seeking for solving techniques with efficiency close to [3] and precision close to [6]. Concretely, we develop a novel technique that breaks down the input *CR* into *atomic CRs* of simpler form, solves each of them separately, and then combines the results into an *UB* for the original *CR*. Our main observation is that it is enough to solve few atomic *CRs* precisely, while solving the others as in [3], without affecting the overall precision. We also propose several methods for precisely solving atomic *CRs*, which are based on the idea of specifying the cost using universally quantified formulas as in [6]. However, we do not require the user to provide any template, and, importantly, the generated formulas have almost a linear form for which quantifier elimination can be done efficiently. Our prototype implementation and experiments [1] demonstrate the effectiveness of this approach.

This paper is organised as follows. Sec. 2 provides the required background on *CRs*. Sec. 3 is the technical core of the paper. Sec. 4 describes a prototype implementation and preliminary experiments. Finally, in Sec. 5 we conclude and discuss related work.

2 Cost Relations: Syntax and Semantics

In this section we recall some basic notions related to *CRs* [3]. The sets of real, rational, and integer values are denoted by $\mathbb{R}$, $\mathbb{Q}$, and $\mathbb{Z}$, respectively. $\mathbb{R}^+$, $\mathbb{Q}^+$, and $\mathbb{Z}^+$ denote their non-negative subsets. Variables are denoted by x, y, z, and

w, possibly subscripted. Values from $\mathbb{R}$, $\mathbb{Q}$, and $\mathbb{Z}$ are denoted, respectively, by r, q, and v. A sequence of elements of type t is denoted by $\bar{t}$. The set of variables of t is denoted by $vars(t)$. An assignment $\sigma : \mathcal{V} \mapsto \mathcal{D}$ maps variables from $\mathcal{V}$ to values from $\mathcal{D}$ and $\sigma(\bar{t})$ denotes the replacement of any $x \in vars(\bar{t})$ by $\sigma(x)$.

A *linear expression* has the form $q_0{+}q_1{\cdot}x_1{+}\cdots{+}q_n{\cdot}x_n$. A *linear constraint* has the form $l_1{\leq}l_2$, $l_1{=}l_2$, or $l_1{\geq}l_2$, where l_1 and l_2 are linear expressions and $vars(l_1) \cup vars(l_2) \subseteq \mathbb{Z}$. The constraints $l_1{>}l_2$ and $l_1{<}l_2$ abbreviate $l_1{\geq}l_2{+}1$ and $l_1{+}1{\leq}l_2$, respectively. We use φ, ϕ, and ψ, possibly subscripted, to denote conjunctions (often written as sets) of linear constraints. We say that φ is *satisfiable* if there is an assignment σ for $vars(\varphi)$ such that $\sigma(\varphi)$ is true, denoted as $\sigma \models \varphi$. If $\sigma \models \varphi$ for every assignment σ for $vars(\varphi)$ then φ is a *valid* formula.

Definition 1 (cost expression). *A* cost expression e *is defined as:*

$$e ::= q \mid \mathsf{nat}(l) \mid \log_a(1{+}\mathsf{nat}(l)) \mid a^{\mathsf{nat}(l)}{-}1 \mid e{+}e \mid e{\cdot}e$$

where $q \in \mathbb{Q}^+$, $\mathsf{nat}(l) = \max\{l, 0\}$, $a > 1 \in \mathbb{Z}^+$, *and* l *is a linear expression.*

Note that we use $a^{\mathsf{nat}(l)}{-}1$, instead of simply $a^{\mathsf{nat}(l)}$, for the sake of simplifying the formal presentation (we explain this after Lemma 3).

Definition 2 (cost relation). *A* cost relation *is a set of* cost equations *of the form* $\langle C(\bar{x}) = e + \sum_{j=1}^{k} D_j(\bar{y}_j), \varphi\rangle$, *where* C *and* D_j *are cost relation symbols.*

Intuitively, a cost equation $\langle C(\bar{x}) = e{+}\sum_{j=1}^{k} D_j(\bar{y}_j), \varphi\rangle$ states that the cost of $C(\bar{x})$ is e plus the sum of the costs of $D_1(\bar{y}_1), \ldots, D_k(\bar{y}_k)$. The linear constraint φ specifies the values of $\bar{x}$ for which the equation is applicable, and defines relations among the different variables. Since *CRs* usually originate from programs, it is often helpful to think of each *CR* symbol as a (non-deterministic) procedure, in which case we say that C calls $D_1, \ldots, D_k$.

Without loss of generality, in what follows we assume that the input *CR* includes a single *CR* symbol. Namely, in Def. 2 we have $D_j{=}C$. We call such *CRs stand-alone*. To handle *CRs* with more than one *CR* symbol, we rely on the compositional approach of [3] which we briefly explain next. In a first step, the input *CR* is transformed into a form in which all recursions are direct, i.e. an equation that defines C can either call itself directly, or other *CR* symbols that do not call C (directly or indirectly). In a second step, the *CRs* are solved iteratively, where in each iteration we solve those that do not depend on any other symbols (there must be at least one), and then substitute the result in the calling contexts. In the rest of the paper *CR* refers to a stand-alone *CR*.

To define the cost assigned by C to a concrete input $\bar{v}$, we use evaluation trees. A (possibly infinite) tree will be denoted by $node(r, \langle T_1, \ldots, T_k\rangle)$, where $r \in \mathbb{R}^+$ is the value of the root and $T_1, \ldots, T_k$ are sub-trees.

Definition 3 (evaluation tree). *Given a* CR C *and an input* $\bar{v}$, *we say that* $node(r, \langle T_1, \ldots, T_k\rangle)$ *is an* evaluation tree *for* $C(\bar{v})$ *iff there exists an equation* $\mathcal{E} \equiv \langle C(\bar{x}) = e{+}\sum_{j=1}^{k} C(\bar{y}_j), \varphi\rangle$ *and* $\sigma : vars(\mathcal{E}) \mapsto \mathbb{Z}$ *such that: (1)* $\sigma(x_i){=}v_i$ *and* $\sigma{\models}\varphi$; *(2)* $r = \sigma(e)$; *and (3) each* T_i *is an evaluation tree for* $C(\sigma(\bar{y}_i))$.

Intuitively, when viewing C as a procedure, an evaluation tree can be seen as a *recursion tree* where the call $C(\bar{v})$ is evaluated as follows: we pick an equation that defines C and an assignment σ that satisfies the equation's constraints; we evaluate $\sigma(e)$ into r, and we recursively call each $C(\sigma(\bar{y}_i))$. Note that an evaluation tree can be infinite. Note also that $C(\bar{v})$ might have several evaluation trees, due to the nondeterminism induced by choosing an equation for C and a satisfying assignment σ for φ. The set of all evaluation trees for $C(\bar{v})$ is denoted by $\mathit{Trees}(C(\bar{v}))$. The set of all possible costs for $C(\bar{v})$ is then defined as $\mathit{Answers}(C(\bar{v}))=\{\mathsf{Sum}(T) \mid T\in\mathit{Trees}(C(\bar{v}))\}$, where $\mathsf{Sum}(T)$ is the sum of all nodes of T. Our interest is to approximate *CRs* by mean of closed-form *UBs* functions, i.e., functions of the form $f(\bar{x})=e$, where $\mathit{vars}(e)\subseteq\bar{x}$.

Definition 4 (upper bound). *A function $C^+:\mathbb{Z}^n\mapsto\mathbb{R}^+$ is an* UB *for a* CR *C, iff for any input $\bar{v}\in\mathbb{Z}^n$ and cost $r\in\mathit{Answers}(C(\bar{v}))$ we have $C^+(\bar{v})\geq r$.*

Next we overview the approach of [3] for solving a *CR* into an *UB*. Suppose we have two functions $h(\bar{x})=e_1$ and $g(\bar{x})=e_2$, where e_1 and e_2 are cost expressions, such that for any $T\in\mathit{Trees}(C(\bar{v}))$ the following holds (i) $h(\bar{v})$ is an *UB* on the depth of T; and (ii) $g(\bar{v})$ is an *UB* on the value of any node of T. Now assuming that d is the maximum number of recursive calls in any equation of C, i.e., the maximum branching factor of its evaluation trees, then $C^+(\bar{x})=g(\bar{x})\cdot\mathcal{N}$ where $\mathcal{N}=h(\bar{x})$ if $d=1$, and $\mathcal{N}=d^{h(\bar{x})}$ if $d>1$. Technically, in [3], $h(\bar{x})$ is computed by inferring a linear *ranking function* [8] that bounds the recursion depth of C, and $g(\bar{x})$ is computed by relying on *linear invariants*.

Example 2. Consider the *CR for* in Fig. 1. The technique of [3] infers $h(e,d,s,i)=\mathsf{nat}(e-i)$ and $g(e,d,s,i)=2+2\cdot\mathsf{nat}(e+s-1)$. Then, since the branching factor is $d=1$, it reports the *UB* $\mathit{for}^+(e,d,s,i)=\mathsf{nat}(e-i)\cdot(2+2\cdot\mathsf{nat}(e+s-1))$. For *CR qs*, it infers $h(f,t,r)=\mathsf{nat}(t-f)$ and $g(f,t,r)=\mathsf{nat}(t-f)^2$. Then, since the branching factor is $d=2$, it reports the *UB* $qs^+(f,t,r)=\mathsf{nat}(t-f)^2\cdot 2^{\mathsf{nat}(t-f)}$. □

Maximisation procedure. We rely on the technique of [3] that generates $g(\bar{x})$ as we explain next. Let e be the cost expression that is contributed by an equation of C, and let b be a cost sub-expression of e. As explained in Def. 3, when generating the nodes of an evaluation tree $T\in\mathit{Trees}(C(\bar{v}))$, we evaluate $\sigma(e)$ to r. This evaluation requires computing $\sigma(b)$. We call $\sigma(b)$ an instance of b. We reuse the techniques of [3] to infer a cost expression $\hat{b}(\bar{x})$ that satisfies the following: for any input $\bar{v}$, $T\in\mathit{Trees}(C(\bar{v}))$ and any instance $\sigma(b)$ of b in T, we have $\hat{b}(\bar{v})\geq\sigma(b)$. Intuitively, $\hat{b}(\bar{x})$ is a function that bounds each contribution of b to the total cost. We call $\hat{b}(\bar{x})$ the *maximisation* of b, and, in our implementation, we compute it reusing the components of [3].

3 Solving Cost Relations in Closed-Form Upper-Bounds

In this section we present our approach for solving a *CR* C into an *UB*. We assume that C is defined by m equations of the form $\langle C(\bar{x})=e_i+\sum_{j=1}^{k_i}C(\bar{y}_{ij}),\varphi_i\rangle$,

for_1	$for(e,d,s,i) = 0$	$\varphi_1 = \{i \geq e\}$
	$for(e,d,s,i) = 2 \cdot \mathsf{nat}(s) + for(e,d',s',i')$	$\varphi_2 = \{i < e, s = d, d' = 2 \cdot d, s' = s+1, i' = i+1\}$
	$for(e,d,s,i) = for(e,d,s',i')$	$\varphi_3 = \{i < e, d > s, s' = s+1, i' = i+1\}$
for_2	$for(e,d,s,i) = 0$	$\varphi_1 = \{i \geq e\}$
	$for(e,d,s,i) = 2 + for(e,d',s',i')$	$\varphi_2 = \{i < e, s = d, d' = 2 \cdot d, s' = s+1, i' = i+1\}$
	$for(e,d,s,i) = for(e,d,s',i')$	$\varphi_3 = \{i < e, d > s, s' = s+1, i' = i+1\}$
for_3	$for(e,d,s,i) = 0$	$\varphi_1 = \{i \geq e\}$
	$for(e,d,s,i) = for(e,d',s',i')$	$\varphi_2 = \{i < e, s = d, d' = 2 \cdot d, s' = s+1, i' = i+1\}$
	$for(e,d,s,i) = 2 + for(e,d,s',i')$	$\varphi_3 = \{i < e, d > s, s' = s+1, i' = i+1\}$
qs_1	$qs(f,t,r) = \mathsf{nat}(t-f)^2$	$\psi_1 = \{t - f < r, r \geq 0\}$
	$qs(f,t,r) = qs(f,m',r) + qs(m'',t,r)$	$\psi_2 = \{t - f \geq r, r \geq 0, f \leq m \leq t,$ $m' = m-1, m'' = m+1\}$
qs_2	$qs(f,t,r) = 0$	$\psi_1 = \{t - f < r, r \geq 0\}$
	$qs(f,t,r) = \mathsf{nat}(t-f) + qs(f,m',r) +$ $qs(m'',t,r)$	$\psi_2 = \{t - f \geq r, r \geq 0, f \leq m \leq t,$ $m' = m-1, m'' = m+1\}$

Fig. 2. The sparse *CRs* of *for* and *qs* of Fig. 1

$1 \leq i \leq m$. Our approach is presented in two steps: we reduce the problem of solving C to solving *atomic CRs*, and then we focus on solving atomic *CRs*.

Observe that cost expressions, as in Def. 1, can be normalised into the form $P_1 + \cdots + P_h$, where each P_i is a *product* of cost expressions $b_{i1}, \ldots, b_{ip_i}$ with $b_{ij} \in \{q, \mathsf{nat}(l), \log_a(1+\mathsf{nat}(l)), a^{\mathsf{nat}(l)} - 1\}$. For simplicity, since q is non-negative, we assume it is given as $\mathsf{nat}(q)$. We assume that each e_i in C is given in this form. Let $P_C = \{P_1, \ldots, P_t\}$ be the multiset of all non-zero product cost expressions that appear in C (i.e., the products of $e_1, \ldots, e_m$). We define C_i as the *CR* obtained from C by removing all $P_j \in P_C$ with $j \neq i$. Namely, in C_i there is exactly one equation that contributes P_i, the others contribute 0. We call such *CRs sparse* and the equation that includes P_i is called the *main equation*.

Example 3. Consider the *CRs for* and *qs* in Fig. 1. Their products are respectively $P_{for} = \{2 \cdot \mathsf{nat}(s), 2, 2\}$ and $P_{qs} = \{\mathsf{nat}(t-f) \cdot \mathsf{nat}(t-f), \mathsf{nat}(t-f)\}$. Their corresponding sparse *CRs* are depicted in Fig. 2. □

Observation 1 *If $C_i^+(\bar{x})$ is an* UB *for the sparse* CR *C_i, for all $1 \leq i \leq t$, then $C^+(\bar{x}) = C_1^+(\bar{x}) + \cdots + C_t^+(\bar{x})$ is an* UB *for C.*

The above observation explains how an *UB* for C can be obtained from *UBs* for its sparse *CRs* $C_1, \ldots, C_t$. Thus, we can focus on solving sparse *CRs*. We first explain the idea intuitively. Assume that $b_{i1} \cdot b_{i2}$ is the product in the main equation of C_i. Given an arbitrary $T \in Trees(C_i(\bar{v}))$, the cost of each of its nodes is either 0 or an instance of $b_{i1} \cdot b_{i2}$. Let $\sigma_1(b_{i1} \cdot b_{i2}), \ldots, \sigma_h(b_{i1} \cdot b_{i2})$ be the instances of $b_{i1} \cdot b_{i2}$ in $T \in Trees(C_i(\bar{v}))$, then the cost of T is $S = \sum_{j=1}^{h} \sigma_j(b_{i1} \cdot b_{i2})$. As explained in Sec. 2, we can compute a function $\hat{b}_{i1}(\bar{x})$ such that $\hat{b}_{i1}(\bar{v}) \geq \sigma_j(b_{i1})$ for each $1 \leq j \leq h$. Using $\hat{b}_{i1}(\bar{v})$ we bound S as follows:

$S = \sum_{j=1}^{h} \sigma_j(b_{i1} \cdot b_{i2}) \leq \sum_{j=1}^{h} \hat{b}_{i1}(\bar{v}) \cdot \sigma_j(b_{i2}) = \hat{b}_{i1}(\bar{v}) \cdot \sum_{j=1}^{h} \sigma_j(b_{i2})$

Now assume that we have a function $f^+(\bar{x})$ such that $f^+(\bar{v}) \geq \sum_{j=1}^{h} \sigma_j(b_{i2})$, then $S \leq \hat{b}_{i1}(\bar{v}) \cdot f^+(\bar{v})$. Thus, since the above reasoning is done for an arbitrary T, we can conclude that $\hat{b}_{i1}(\bar{x}) \cdot f^+(\bar{x})$ is an *UB* for C_i. Now to compute $f^+(\bar{x})$, we consider a *CR* C_{i2} that is obtained from C_i by replacing $b_{i1} \cdot b_{i2}$ by b_{i2}. Clearly, $C_{i2}(\bar{v}) = \sum_{j=1}^{h} \sigma_j(b_{i2})$, and thus any *UB* for C_{i2} defines a valid $f^+(\bar{x})$. This reduces the problem of solving C_i to that of solving C_{i2}, which is simpler since its main equation includes a basic cost expression. Note that, in a similar way, we could build C_{i1} using $\hat{b}_{i2}(\bar{x})$ and then use it to find an *UB* for C_i.

Formally, given a sparse *CR* C_i with a product $b_{i1} \cdot \cdots \cdot b_{ip_i}$ in its main equation, we define the *atomic CR* C_{ij} as the one obtained from C_i by replacing its product by b_{ij} (i.e., removing all b_{ik} with $k \neq j$).

Example 4. Consider the sparse *CRs* depicted in Fig. 2. The following are possible atomic *CRs* for for_1 and qs_1

for_{12}	
$for(e,d,s,i)=0$	φ_1
$for(e,d,s,i)=\mathsf{nat}(s)+for(e,d',s',i')$	φ_2
$for(e,d,s,i)=for(e,d,s',i')$	φ_3

qs_{11}	
$qs(f,t,r)=\mathsf{nat}(t-f)$	ψ_1
$qs(f,t,r)=qs(f,m',r)+qs(m'',t,r)$	ψ_2

in which $\mathsf{nat}(s)$ and $\mathsf{nat}(t-f)$ are selected as basic cost expressions. *CRs* for_2, for_3 and qs_2 are already atomic. They correspond to for_{21}, for_{31} and qs_{21}. □

Lemma 1. *Let* C_i *be a* sparse CR, $b_{i1} \cdot \cdots \cdot b_{ip_i}$ *the product in its main equation, and* C_{ij} *an atomic* CR *of* C_i. *If* $C^+_{ij}(\bar{x})$ *is an* UB *for* $C_{ij}(\bar{x})$, *then* $C^+_i(\bar{x}) = C^+_{ij}(\bar{x}) \cdot \prod_{k \neq j} \hat{b}_{ik}(\bar{x})$ *is an* UB *for* C_i.

The above lemma allows focusing on finding an *UB* for a single atomic C_{ij} and then combine the result into an *UB* for C_i. To put this into practice we need to address the following issues: (1) how to select the basic cost expression j from the products in order to build C_{ij}; and (2) how to compute an *UB* for C_{ij}. In secs. 3.1 and 3.2 we discuss several methods for addressing the second issue. The first issue is discussed later in Sec. 3.4.

Let us first position our approach in the spectrum of related approaches [3,6]. Solving C_i using the techniques of [3] we obtain the *UB* $(\prod_{k=1}^{p_i} \hat{b}_{ik}(\bar{x})) \cdot \mathcal{N}$. Interestingly, this *UB* can be explained using our novel view of Lemma 1, which is different from that of [3], as follows: we can consider $\hat{b}_{ij}(\bar{x}) \cdot \mathcal{N}$ as an *UB* for C_{ij}, and then use it as in Lemma 1 to obtain $(\prod_{k=1}^{p_i} \hat{b}_{ik}(\bar{x})) \cdot \mathcal{N}$. Since, unlike [3], we focus on solving atomic *CRs*, we develop dedicated techniques (i.e., techniques that work only for atomic *CRs*) that are able to obtain an *UB* far more precise than $\hat{b}_{ij}(\bar{x}) \cdot \mathcal{N}$ (we will usually eliminate the $\mathcal{N}$ factor). Solving C_i using the techniques of [6] requires defining an *UB* template to be used during the solving process. If C_i does not admit an *UB* that matches the supplied templates, then this technique will fail. Moreover, using arbitrary templates renders this approach impractical since it is based on the use of quantifier elimination procedure. Our techniques for solving atomic *CRs* are actually inspired by those

of [6]. However, since we focus on a simpler form of *CRs*, we always use linear templates for which the quantifier elimination procedure is efficient. In summary, our approach uses [6] to precisely reason on the *local* cost of a single simple cost expression b_{ij}, and then uses [3] to combine this local cost into an *UB* for C_i.

To simplify our notation, in what follows, we assume a given atomic *CR* D with m equations of the form $\langle D(\bar{x}) = e_i + \sum_{j=1}^{k_i} D(\bar{y}_{ij}), \varphi_i \rangle$, where $e_1 = b$ is a basic cost expression, and $e_i = 0$ for all $2 \leq i \leq m$. Note that the main equation of D is the first one. We denote by $\bar{w}_i$ the set of variables in the i-th equation.

3.1 The Tree-Sum Method

We first explain this method for the case in which $b = \mathsf{nat}(l)$, and then we show how to extend it to handle any basic cost expression b. In many cases, in particular in examples that require amortised analysis, the sum of all instances of b in any $T \in \mathit{Trees}(D(\bar{v}))$ can be bounded by a linear expression. Thus, we seek an *UB* for D of the form $\alpha(\bar{x}) = q_0 + q_1 \cdot x_1 + \cdots + q_n \cdot x_n$, where $q_i \in \mathbb{Q}$. The way we search for $\alpha(\bar{x})$ is based on the use of universally quantified formulas as in [6]. We first define a verification condition which ensures that a given $\alpha(\bar{x})$ is a valid *UB* for D. Then, using a quantifier elimination procedure, we turn this verification condition into a synthesis procedure that actually infers $\alpha(\bar{x})$.

Lemma 2. *Let* $\alpha(\bar{x}) = q_0 + q_1 \cdot x_1 + \cdots + q_n \cdot x_n$*, and define:*

$$\Psi_1 \triangleq \forall \bar{w}_1 \ : \ \varphi_1 \rightarrow \mathsf{nat}(\alpha(\bar{x})) \geq \mathsf{nat}(l) + \textstyle\sum_{j=1}^{k_1} \mathsf{nat}(\alpha(\bar{y}_{1j}))$$
$$\Psi_2 \triangleq \textstyle\bigwedge_{i=2}^{m} \forall \bar{w}_i \ : \ \varphi_i \rightarrow \mathsf{nat}(\alpha(\bar{x})) \geq \sum_{j=1}^{k_i} \mathsf{nat}(\alpha(\bar{y}_{ij}))$$

If $\Psi_1 \wedge \Psi_2$ *is valid, then* $\mathsf{nat}(\alpha(\bar{x}))$ *is an* UB *for the atomic* CR D*.*

Intuitively, Ψ_1 requires that $\mathsf{nat}(\alpha(\bar{x}))$ covers the cost of the main equation, i.e., it covers the local cost $\mathsf{nat}(l)$ and the cost of the recursive calls. Similarly, Ψ_2 requires that $\mathsf{nat}(\alpha(\bar{x}))$ covers the cost of the other equations (in this case the local cost is 0). Our main interest is in inferring such $\alpha(\bar{x})$ rather than verifying the correctness of a given one. Turning the verification condition into an inference procedure can be done, using a quantifier elimination procedure, as follows:

1. we generate $\Psi_1 \wedge \Psi_2$ using a *template* function $\alpha(\bar{x})$ in which $q_0, \ldots, q_n$ are variables, i.e., *unknown*;
2. we eliminate the universal quantifiers from $\Psi_1 \wedge \Psi_2$. This results in a set of constraints Θ over the variables $q_0, \ldots, q_n$; and
3. any solution of Θ (i.e., values for $q_0, \ldots, q_n$ that satisfy Θ) defines a valid *UB* $\mathsf{nat}(\alpha(\bar{x}))$. We simply pick a solution.

Note that if Θ is not satisfiable then there is no $\alpha(\bar{x})$ satisfying $\Psi_1 \wedge \Psi_2$. In such case we say that the Tree-Sum method is not applicable for D. The main subtle point in the above inference procedure is how to eliminate the universal quantifiers, which is computationally expensive in general. However, since the formula $\Psi_1 \wedge \Psi_2$ have a very specific form (almost linear), in Sec. 3.3 we show how this can be done efficiently. For now we just assume the existence of a procedure that implements steps (2) and (3) above.

Example 5. Consider the *CR* for_{12}, as defined in Ex. 4, and let $\alpha(e,d,s,i) = q_0 + q_1{\cdot}e + q_2{\cdot}d + q_3{\cdot}s + q_4{\cdot}i$. The corresponding Ψ_1 and Ψ_2 are:

$$\Psi_1 \triangleq \forall \bar{w}_2 : \varphi_2 \rightarrow \mathsf{nat}(q_0+q_1{\cdot}e+q_2{\cdot}d+q_3{\cdot}s+q_4{\cdot}i) \geq \mathsf{nat}(s)+\mathsf{nat}(q_0+q_1{\cdot}e+q_2{\cdot}d'+q_3{\cdot}s'+q_4{\cdot}i')$$
$$\Psi_2 \triangleq \forall \bar{w}_3 : \varphi_3 \rightarrow \mathsf{nat}(q_0+q_1{\cdot}e+q_2{\cdot}d+q_3{\cdot}s+q_4{\cdot}i) \geq \mathsf{nat}(q_0+q_1{\cdot}e+q_2{\cdot}d+q_3{\cdot}s'+q_4{\cdot}i')$$

Solving $\Psi_1 \wedge \Psi_2$, i.e finding values for $q_0, \ldots, q_4$, gets $q_0{=}{-}2$, $q_1{=}2$, $q_2{=}{-}1$, $q_3{=}2$, and $q_4{=}{-}2$, which means that $for^+_{12}(e,d,s,i){=}\mathsf{nat}(2{\cdot}s{+}2{\cdot}e{-}2{\cdot}i{-}d{-}2)$ is an *UB* for *CR* for_{12}. Then, to get an *UB* for *CR* for_1 we apply Lemma 1 which results in $for^+_1(e,d,s,i){=}2{\cdot}\mathsf{nat}(2{\cdot}s{+}2{\cdot}e{-}2{\cdot}i{-}d{-}2)$. Similarly, generating the formulas for for_{21} and for_{31} and solving them, we get the *UBs* $for^+_2(e,d,s,i){=}\mathsf{nat}(2{\cdot}e{-}2{\cdot}i)$ and $for^+_3(e,d,s,i){=}\mathsf{nat}(2{\cdot}e{-}2{\cdot}i)$. Finally, we can use Obs. 1 to add them in $for^+(e,d,s,i){=}2{\cdot}\mathsf{nat}(2{\cdot}s{+}2{\cdot}e{-}2{\cdot}i{-}d{-}2){+}2{\cdot}\mathsf{nat}(2{\cdot}e{-}2{\cdot}i)$ as *UB* for *for*. Substituting this *UB* in the equation of *add* in Fig. 1, we get the expected linear bound $add^+(e,d,s) = 2{\cdot}\mathsf{nat}(2{\cdot}s{+}2{\cdot}e{-}d{-}2){+}2{\cdot}\mathsf{nat}(2{\cdot}e)$ for method `add`. □

Now we turn to the general case in which b is an arbitrary basic cost expression, not necessarily $\mathsf{nat}(l)$. In such cases, in addition to $\mathsf{nat}(l)$, b can be of the form $\log_a(1+\mathsf{nat}(l))$ or $a^{\mathsf{nat}(l)}-1$. Recall that when it is $q \in \mathbb{Q}^+$, we have implicitly assumed it was written as $\mathsf{nat}(q)$. Note that in all cases b has an embedded $\mathsf{nat}(l)$ expression. Let E be the *CR* obtained from D by replacing b by its embedded $\mathsf{nat}(l)$. Then the following lemma explains how to obtain an *UB* for D from that of E. Computing an *UB* for E is done as above.

Lemma 3. *Let* $\mathsf{nat}(\alpha(\bar{x}))$ *be an* UB *for* E*, and let*

$$D^+(\bar{x}) = \begin{cases} \mathsf{nat}(\alpha(\bar{x})) & b = \mathsf{nat}(l) \\ 1.5{\cdot}\mathsf{nat}(\alpha(\bar{x})) & b = \log_a(1+\mathsf{nat}(l)) \\ a^{\mathsf{nat}(\alpha(\bar{x}))}-1 & b = a^{\mathsf{nat}(l)}-1 \end{cases}$$

Then, $D^+(\bar{x})$ *is an* UB *for* D*.*

It is worth mentioning here the reason for which we use $a^{\mathsf{nat}(l)}-1$ as a basic cost expression, instead of $a^{\mathsf{nat}(l)}$. This allows *precisely* lifting the *UB* of E to an *UB* of D (in the last case of D^+), which is not possible when using $a^{\mathsf{nat}(l)}$.

Example 6. Let us finish this section by trying to analyse the *CR* *qs* using the Tree-Sum method. For qs_{11}, we first generate:

$$\Psi_1 \triangleq \forall \bar{w}_1 \ : \ \psi_1 \rightarrow \mathsf{nat}(q_0+q_1{\cdot}f+q_2{\cdot}t+q_3{\cdot}r) \geq \mathsf{nat}(t-f)$$
$$\Psi_2 \triangleq \forall \bar{w}_2 \ : \ \psi_2 \rightarrow \mathsf{nat}(q_0+q_1{\cdot}f+q_2{\cdot}t+q_3{\cdot}r) \geq \mathsf{nat}(q_0+q_1{\cdot}f+q_2{\cdot}m'+q_3{\cdot}r){+}\mathsf{nat}(q_0+q_1{\cdot}m''+q_2{\cdot}t+q_3{\cdot}r)$$

Solving $\Psi_1 \wedge \Psi_2$ results in $q_0 = 0$, $q_1 = -1$, $q_2 = 1$, and $q_3 = 0$. Thus, $\mathsf{nat}(t-f)$ is an *UB* for qs_{11}. Using Lemma 1, we get $qs^+_1(f,t,r){=}\mathsf{nat}(t-f)^2$. Solving qs_{21} with the Tree-Sum method does not yield any result because the generated formula is not valid. This is expected since qs_{21} does not have a linear bound. In Sec. 3.2 we develop further methods to handle such cases. □

3.2 The Level-Sum Method

In this section we describe our method for solving atomic *CRs* that exhibit a *divide and conquer* like behaviour. As we have seen in Ex. 6, the Tree-Sum method fails to handle such examples. We first explain it for the case of $b = \mathsf{nat}(l)$, and then extend it to an arbitrary basic cost expression.

We start with some notation. Given an evaluation tree $T \in \mathit{Trees}(D(\bar{v}))$, a node in T is called *primary* if it is generated by the main equation. Note that the cost of all other nodes in T is 0. The *primary-depth* of a primary node is the number of primary nodes on the path from the root to that node (both included). The primary-depth of T, denoted by $\mathit{pdepth}(T)$, is the maximum among the primary depths of all its primary nodes. The sum of (the cost of) all primary nodes of primary-depth i is denoted by $\mathsf{SumLevel}(T, i)$.

We say that $\mathsf{nat}(\alpha(\bar{x}))$ is an UB *on the primary-depth* of D, if for any input $\bar{v}$ and $T \in \mathit{Trees}(D(\bar{v}))$ we have $\mathsf{nat}(\alpha(\bar{v})) \geq \mathit{pdepth}(T)$. We say that it is an UB *on the Level-Sum* of D, if for any input $\bar{v}$, $T \in \mathit{Trees}(D(\bar{v}))$, and $1 \leq i \leq \mathit{pdepth}(T)$ we have $\mathsf{nat}(\alpha(\bar{v})) \geq \mathsf{SumLevel}(T, i)$.

Lemma 4. *Let* $\mathsf{nat}(\alpha_1(\bar{x}))$ *and* $\mathsf{nat}(\alpha_2(\bar{x}))$ *be* UBs *on the primary-depth and Level-Sum of* D*, respectively. Then,* $\mathsf{nat}(\alpha_1(\bar{x}))\cdot\mathsf{nat}(\alpha_2(\bar{x}))$ *is an* UB *for* D*.*

The correctness of the above lemma follows from the fact that only primary nodes can have non-zero cost. Intuitively, the above lemma handles divide and conquer examples since, in such examples, the input is distributed between the recursive calls. Thus, the cost of all levels is similar and can be expressed as a linear function on the initial input. Moreover, using the primary-depth, instead of depth, allows ignoring those levels that do not contribute to the cost. Note that the above lemma also reduces the problem of solving D, to that of finding $\mathsf{nat}(\alpha_1(\bar{x}))$ and $\mathsf{nat}(\alpha_2(\bar{x}))$ that bound its primary-depth and Level-Sum. We start with bounding the primary-depth.

Lemma 5. *Let* $\alpha(\bar{x}) = q_0 + q_1\cdot x_1 + \cdots + q_n \cdot x_n$*, and define:*

$$\Phi_1 \triangleq \begin{cases} \forall \bar{w}_1 \ : \ \varphi_1 \rightarrow \mathsf{nat}(\alpha(\bar{x})) \geq 1 & \text{if } k_1 = 0 \\ \bigwedge_{j=1}^{k_1} \forall \bar{w}_1 \ : \ \varphi_1 \rightarrow \mathsf{nat}(\alpha(\bar{x})) \geq 1 + \mathsf{nat}(\alpha(\bar{y}_{1j})) & \text{if } k_1 \geq 1 \end{cases}$$

$$\Phi_2 \triangleq \bigwedge_{i=2}^{m} \bigwedge_{j=1}^{k_i} \forall \bar{w}_i \ : \ \varphi_i \rightarrow \mathsf{nat}(\alpha(\bar{x})) \geq \mathsf{nat}(\alpha(\bar{y}_{ij}))$$

If $\Phi_1 \wedge \Phi_2$ *is valid, then* $\mathsf{nat}(\alpha(\bar{x}))$ *is an* UB *on the primary-depth of* D*.*

Intuitively, the primary-depth corresponds to the number of applications of the main equation, in a sequence of recursive calls. This is reflected in Φ_1 and Φ_2 as follows. In Φ_1, we treat applications of the main equation. If the main equation is non-recursive, i.e., $k_1 = 0$, then we require that $\mathsf{nat}(\alpha(\bar{x}))$ covers that single application. In case it is recursive, i.e., $k_1 \geq 1$, then we require that $\mathsf{nat}(\alpha(\bar{x}))$ covers that application and further ones that might arise through each recursive call. In Φ_2, we treat applications of other equations. In such case we require that $\mathsf{nat}(\alpha(\bar{x}))$ covers applications of the main equation that might arise through each recursive call. Note that each recursive call is considered separately, since we count primary nodes in each path rather than the whole tree.

It is worth noting that if we apply Φ_1 to all equations instead of only the main one, then $\mathsf{nat}(\alpha(\bar{x}))$ bounds the depth of any evaluation tree rather than the primary-depth. Similar techniques, based on inference of (linear) ranking functions, were used in [3] to bound the depth of the evaluation trees.

Example 7. Applying Lemma 5 to bound the primary-depth of qs_{21} (of Ex. 4) results in $\Phi_2 = \mathit{true}$ and Φ_1 as the conjunction of the following formulas:

$$\forall \bar{w}_2 \ : \ \psi_2 \rightarrow \mathsf{nat}(q_0{+}q_1{\cdot}f{+}q_2{\cdot}t{+}q_3{\cdot}r) \geq 1{+}\mathsf{nat}(q_0{+}q_1{\cdot}f{+}q_2{\cdot}m'{+}q_3{\cdot}r)$$
$$\forall \bar{w}_2 \ : \ \psi_2 \rightarrow \mathsf{nat}(q_0{+}q_1{\cdot}f{+}q_2{\cdot}t{+}q_3{\cdot}r) \geq 1{+}\mathsf{nat}(q_0{+}q_1{\cdot}m''{+}q_2{\cdot}t{+}q_3{\cdot}r)$$

Both originate from the recursive equation of qs_2. They respectively correspond to the first and second calls. Solving $\Phi_1 \wedge \Phi_2$ results in $q_0 = 1$, $q_1 = -1$, $q_2 = 1$, $q_3 = 0$, which induces the *UB* $\mathsf{nat}(t{-}f{+}1)$ on the primary-depth of qs_{21}. □

Now we turn to bounding the Level-Sum of D.

Lemma 6. *Let* $\alpha(\bar{x}) = q_0{+}q_1{\cdot}x_1{+}\cdots{+}q_n{\cdot}x_n$, *and define:*

$$\Pi_1 \triangleq \forall \bar{w}_1 : \varphi_1 \rightarrow \quad \mathsf{nat}(\alpha(\bar{x})) \geq \mathsf{nat}(l)$$
$$\Pi_2 \triangleq \textstyle\bigwedge_{i=1}^{m} \forall \bar{w}_i : \varphi_i \rightarrow \quad \mathsf{nat}(\alpha(\bar{x})) \geq \sum_{j=1}^{k_i} \mathsf{nat}(\alpha(\bar{y}_{ij}))$$

If $\Pi_1 \wedge \Pi_2$ *is valid, then* $\mathsf{nat}(\alpha(\bar{x}))$ *is an* UB *on the Level-Sum of* D.

Intuitively, Π_1 requires that $\mathsf{nat}(\alpha(\bar{x}))$ covers the local cost of the main equation at any level, and Π_2 requires that it also covers the next level. Combining these conditions, and applying inductive reasoning, one can conclude that $\mathsf{nat}(\alpha(\bar{x}))$ is actually an *UB* on the Level-Sum of D.

Example 8. Consider again qs_{21} (of Ex. 4). Its corresponding formulas are:

$$\Pi_1 \triangleq \forall \bar{w}_2 \ : \ \psi_2 \rightarrow \mathsf{nat}(q_0{+}q_1{\cdot}f{+}q_2{\cdot}t{+}q_3{\cdot}r) \geq \mathsf{nat}(t{-}f)$$
$$\Pi_2 \triangleq \forall \bar{w}_2 \ : \ \psi_2 \rightarrow \mathsf{nat}(q_0{+}q_1{\cdot}f{+}q_2{\cdot}t{+}q_3{\cdot}r) \geq \mathsf{nat}(q_0{+}q_1{\cdot}f{+}q_2{\cdot}m'{+}q_3{\cdot}r){+}\mathsf{nat}(q_0{+}q_1{\cdot}m''{+}q_2{\cdot}t{+}q_3{\cdot}r)$$

Solving $\Pi_1 \wedge \Pi_2$ results in $q_0{=}0$, $q_1{=}-1$, $q_2{=}1$, $q_3{=}0$. This induces the bound $\mathsf{nat}(t{-}f)$ on the Level-Sum. Combining this bound with that in Ex. 7, on the primary depth, we obtain $\mathsf{nat}(t{-}f){\cdot}\mathsf{nat}(t{-}f{+}1)$ as an *UB* for qs_{21}, which is also an *UB* for qs_2. Combining this, using Obs. 1, with the bound of qs_1 computed in Ex. 6, we get $qs^{+}(f,t,r){=}\mathsf{nat}(t{-}f){\cdot}\mathsf{nat}(t{-}f{+}1){+}\mathsf{nat}(t{-}f){\cdot}\mathsf{nat}(t{-}f)$. Substituting this *UB* in the equation of *qsort* in Fig. 1 we obtain $qsort^{+}(s,r){=}\mathsf{nat}(s{-}1){\cdot}\mathsf{nat}(s){+}\mathsf{nat}(s{-}1){\cdot}\mathsf{nat}(s{-}1)$, which is the expected bound for method qsort. □

Turning the verification condition to inference procedure, both in Lemma 5 and Lemma 6, is done as we explained in Sec. 3.1. Handling the general case in which b is an arbitrary basic cost expression, is done exactly as the case of Tree-Sum (see Lemma 3). Note that this affects only the *UB* on the Level-Sum.

Finally, we note that [3] proposed a technique for solving *CRs* with a divide and conquer behaviour, however, it is limited to cases in which: (1) the cost of all levels is non-increasing; and (2) the cost expression of each equation is linear. Note that, *CR* qs_1, for example, does not satisfy both conditions.

3.3 Solving the Universally Quantified Formulas

In this section we describe how we solve the universally quantified formulas of Lemma 2, Lemma 5, and Lemma 6. Namely, starting from a template linear function $\alpha(\bar{x}) = q_0{+}q_1{\cdot}x_1{+}\ldots{+}q_n{\cdot}x_n$, we find rational values for $q_0, \ldots, q_n$ for which the corresponding formula is valid. Note that our formulas are conjunctions of universally quantified formulas of the following form:

$$\forall \bar{w} : \ \varphi \rightarrow \mathsf{nat}(l_0) \geq q + \mathsf{nat}(l_1) + \ldots + \mathsf{nat}(l_n) \tag{1}$$

where φ defines a closed polyhedron, $q \in \{0, 1\}$, and each l_i is either a linear function over $\bar{w}$, or a template function $\alpha(\bar{x}) = q_0{+}q_1{\cdot}x_1{+}\cdots{+}q_n{\cdot}x_n$ such that $\bar{x} \subseteq \bar{w}$ and $q_i \notin \bar{w}$ (i.e., each q_i is existentially quantified). Our goal is to solve these formulas using linear programming (LP) techniques.

Consider a formula as in (1), but without the nat-expressions, i.e., of the form $\forall \bar{w} : \varphi \rightarrow l_0 \geq q{+}l_1{+}\ldots{+}l_n$. It is known that there is a complete algorithm, based on the use of LP [8], able to solve such a formula. Our aim is to transform formulas as (1) to a nat-free as above, and then solve them using this algorithm. Recall that $\mathsf{nat}(l_i) = \max\{l_i, 0\}$. This means that $\mathsf{nat}(l_i)$ can be eliminated by explicitly considering the cases for $l_i \geq 0$ and $l_i \leq 0$ (we use $l_i \geq 0$ and not $l_i > 0$ since in LP constraints must be non-strict). For example, eliminating $\mathsf{nat}(l_0)$ can be done by rewriting (1) as:

$$\begin{array}{ll} \forall \bar{w} : \ \varphi \wedge l_0 \geq 0 \rightarrow l_0 \geq q + \mathsf{nat}(l_1) + \ldots + \mathsf{nat}(l_n) & \bigwedge \\ \forall \bar{w} : \ \varphi \wedge l_0 \leq 0 \rightarrow 0 \ \geq q + \mathsf{nat}(l_1) + \ldots + \mathsf{nat}(l_n) & \end{array}$$

This process can be applied iteratively to eliminate each $\mathsf{nat}(l_i)$. There is still one problem that prevents us from directly applying the LP techniques: when l_i is a template function, the constraints $l_0 \geq 0$ and $l_0 \leq 0$ are not linear. To overcome this problem, assuming that eliminating the nat-expression results in a formula ξ, we generated ξ' be the by simply removing all non-linear constraints from ξ. Since all non-linear constraints in ξ appear in the left-hand sides of the implications, we observe that $\xi' \rightarrow \xi$. This means that we can solve ξ', using the LP based algorithm, instead of ξ. Although we scarify completeness, this approach performs well in practice as demonstrated by our experiments.

3.4 Concluding Remarks

Let us conclude this section describing how all pieces, that have been described so far, connects together to infer an *UB* for C.

Solving CR C. This is as done according to the following steps: (1) generating the sparse *CRs* $C_1, \ldots, C_t$ of C; (2) solving each C_i into an *UB* as described below; and (3) combining these *UBs*, as in Obs. 1, into an *UB* for C.

Solving a sparse CR C_i. This step requires solving, using the methods described in secs. 3.1 and 3.2, one C_{ij} of the corresponding atomic *CR* which might fail for some j and succeed for some others. We iterate over all possible $j{=}1, \ldots, p_i$, and if all fail then we solve C_i using the approach of [3].

Solving an atomic CR C_{ij}. This is done by trying the methods of secs. 3.1 and 3.2, in this order. Note that in [1] we describe some additional methods.

Table 1. Experimental comparison with PUBS [3]. The times on the right (in secs) correspond to analysing a *CR* that connects all benchmarks together (see Sec. 4).

Entry	$\mathcal{O}(ub)$ – **new**	$\mathcal{O}(ub)$ – PUBS	**Eq**	$\mathbf{T}_n$	$\mathbf{T}_p$	**Ov**
add(a,b,c)	$\mathsf{nat}(a)+\mathsf{nat}(2a-b+2c)$	$\mathsf{nat}(a)\cdot\mathsf{nat}(a+c)$	11	0.15	0.11	1.34
qsort(a,b,c)	$\mathsf{nat}(a)^2$	$2^{\mathsf{nat}(a)}\cdot(\mathsf{nat}(a)+\mathsf{nat}(b))$	28	0.61	0.27	2.26
sum(a)	$\mathsf{nat}(a)$	$2^{\mathsf{nat}(a)}\cdot\mathsf{nat}(a)$	36	0.88	0.33	2.63
dac(a,b)	$\mathsf{nat}(a)^2+\mathsf{nat}(a-b)$	$2^{\mathsf{nat}(b)}\cdot\mathsf{nat}(a)$	45	1.24	0.40	3.13
log(a,b)	$\mathsf{nat}(b)+\mathsf{nat}(a)\cdot\mathsf{log}(\mathsf{nat}(b))$	$\mathsf{nat}(b)\cdot\mathsf{nat}(a)$	54	1.71	0.47	3.63
once(a,b)	$\mathsf{nat}(a)+\mathsf{nat}(b)$	$\mathsf{nat}(b)\cdot\mathsf{nat}(a)$	62	1.98	0.57	3.47
twice(a,b)	$\mathsf{nat}(a)+\mathsf{nat}(b)$	$\mathsf{nat}(b)\cdot\mathsf{nat}(a)$	70	2.29	0.69	3.33
full(a,b)	$\mathsf{nat}(a)\cdot\mathsf{nat}(b)$	$\mathsf{nat}(a)\cdot\mathsf{nat}(b)^2$	78	2.74	0.84	3.26
eratos(a)	$\mathsf{nat}(a)$	$\mathsf{nat}(a)^2$	91	3.16	0.94	3.37
peak(a)	$\mathsf{nat}(a)$	$\mathsf{nat}(a)\cdot\mathsf{log}(\mathsf{nat}(a))$	96	3.43	1.01	3.38
stack(a,b,c)	$\mathsf{nat}(b)\cdot\mathsf{nat}(c)+\mathsf{nat}(b)^2$	$\mathsf{nat}(c)\cdot\mathsf{nat}(b)^2$	107	3.95	1.19	3.32
rotate(a,b)	$\mathsf{nat}(a)+\mathsf{nat}(b)+\mathsf{nat}(a-b)$	$\mathsf{nat}(a)\cdot\mathsf{nat}(a-b)$	120	4.84	1.62	2.99
maxsum(a,b)	$\mathsf{nat}(b)\cdot\mathsf{log}(\mathsf{nat}(b))$	$\mathsf{nat}(b)^2$	138	7.67	2.12	3.62
mayor(a)	$\mathsf{nat}(a)\cdot\mathsf{log}(\mathsf{nat}(a))$	$\mathsf{nat}(a)\cdot\mathsf{log}(\mathsf{nat}(a))$	163	13.21	3.20	4.13
msort(a,b,c,d)	$\mathsf{nat}(d-c)\cdot\mathsf{log}(\mathsf{nat}(d-c))$	$\mathsf{nat}(d-c)^2$	173	13.72	3.81	3.60
mergexp(a)	$\mathsf{nat}(a)$	$\mathsf{nat}(a)\cdot\mathsf{log}(\mathsf{nat}(a))$	187	14.65	4.02	3.65
enque(a,b,c,d)	$\mathsf{nat}(c+d)+\mathsf{nat}(a+c)$	$\mathsf{nat}(c+d)\cdot\mathsf{nat}(a+c)$	199	16.34	4.68	3.49
deque(a,b,c)	$\mathsf{nat}(a)+\mathsf{nat}(c)$	$\mathsf{nat}(c)\cdot\mathsf{nat}(a)$	208	17.40	4.91	3.55
infinity(a)	$\mathsf{nat}(a)$	Failed: No RF	219	18.38	5.07	3.63

4 Implementation and Experiments

We have implemented our techniques as an extension of PUBS [3], the solver used in COSTA [4] for solving *CRs* generated from JAVA programs. This allows us to evaluate our approach directly on JAVA programs. We evaluate accuracy and scalability on a set of benchmarks that we collected from related literature, or were written to demonstrate some powerful features of our approach. Although the programs are not large, they exhibit challenging behaviour for cost analysis. The benchmarks and the implementation are available online [1].

In Table 1 we evaluate the accuracy of our approach by comparing it to PUBS [3]. We applied both approaches on each benchmark using a cost model that measure memory consumption or visits to an specific program point (depending on what was more interesting for each benchmark). Each line includes (from left to right) the entry method and its parameters, the *UB* inferred by our approach and the *UB* inferred by PUBS. For readability, bounds are given in asymptotic form [2]. In all examples our approach obtains *UBs* that are asymptotically more accurate than those obtained by PUBS. Moreover, our *UBs* approach obtains precise asymptotic *UBs*, i.e., they exactly reflect the actual cost.

To analyse scalability, we have merged all our benchmarks into a single program as follows: the benchmark in row i was modified to include a call (in one of its loops) to the program at row $i-1$. This means that the i-th benchmark executes at least i nested loops. The runtime (in seconds) of analysing each such (modifed) benchmark is depicted in columns $\mathbf{T}_n$ (current approach) and

$\mathbf{T}_p$ (PUBS) of Table 1. Columns **Eq** and **Ov** are, respectively, the total number of equations and the overhead ($\mathbf{T}_n/\mathbf{T}_p$) introduced by our approach.

We have also compared our approach to [6]. For all benchmarks of Table 1, it did not obtain an *UB* within the one minute time limit. This is expected since it is based on a general procedure for real quantifier elimination.

5 Conclusions and Related Work

In this paper we have developed a novel approach for solving *CRs* into precise closed-form *UBs*. It is based on the idea of dividing the *basic cost expressions* of a given *CR C* into two parts: (a) those for which we employ precise reasoning to track their behaviour along the execution; and (b) those for which we simply use their worst case behavior. Then, we show how such different bounds can be combined into an *UB* for *C*. For part (b) we rely on existing techniques [4] to *maximise* cost expressions. For part (a) we first model the contribution of the corresponding cost expressions using universally quantified formulas, and then, a precise *UB* on their costs can be obtain by eliminating the universal quantifiers. Note that while quantifier elimination is a very expensive procedure in general, in our case, since the formulas are of a very specific form, they can be solved efficiently. Our method has been implemented within COSTA [4], and preliminary experiments demonstrate its superiority on previous methods for solving *CRs*.

Related work. The most related works to ours are [4,6] which aim at solving *CRs* into closed-form *UBs*. In Sec. 4 we have seen that, in practice, our approach is more precise than [4] and more efficient than [6]. Detailed discussion on similarities and differences is provided along Sec. 3. Note that although the method described so far is usually more precise than [3], as we have seen in Sec. 4, there are some examples for which the use of the last case of Lemma 3 causes a loss of precision. E.g., replacing $\mathsf{nat}(s)$ by $2^{\mathsf{nat}(s)}$ in for_{12} of Ex. 4, the approach of [3] obtains $\mathsf{nat}(e-i)\cdot 2^{\mathsf{nat}(s+e-1)}$ while we obtain $2^{\mathsf{nat}(2(s+e-i-1)-d)}$. In [5], the techniques of [4] were improved to handle cases in which the cost can be modeled with arithmetic or geometric sequences. This approach is complementary to ours, in the sense that it cannot handle our benchmarks and we cannot handle some of their examples (when basic cost expressions require non-linear bounds).

There are some works that aim at inferring loop bounds on the visits to a given program point [14,19]. They are mostly related to our Lemma 5. These approaches are not limited to linear bounds, however, they cannot handle recursive programs with more than one recursive call. Our techniques can benefit from these approaches when each cost equation has at most one recursive call. Cost analysis techniques that are based on *amortised analysis* [15,16], could, in principle, handle some of our examples when the bounds are polynomial, and the data are over the non-negative integers. Solving *CRs* using template functions and real quantifier elimination has been considered before in [7]. Finally, several cost analysis frameworks [9,11] that are based on generating *CRs* can benefit from our advances in solving *CRs*.

Acknowledgements. This work was funded partially by the projects FP7-ICT-610582, TIN2008-05624, TIN2012-38137, PRI-AIBDE-2011-0900 and S2009TIC-1465. Diego Esteban Alonso-Blas is supported by the PhD scholarship program of the Complutense University.

References

1. Companion Web-Page, `http://costa.ls.fi.upm.es/amor/`
2. Albert, E., Alonso, D., Arenas, P., Genaim, S., Puebla, G.: Asymptotic Resource Usage Bounds. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 294–310. Springer, Heidelberg (2009)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-Form Upper Bounds in Static Cost Analysis. J. Autom. Reasoning 46(2), 161–203 (2011)
4. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Object-Oriented Bytecode Programs. Theor. Comput. Sci. 413(1), 142–159 (2012)
5. Albert, E., Genaim, S., Masud, A.N.: On the Inference of Resource Usage Upper and Lower Bounds. ACM Trans. Comput. Log. (to appear, 2013)
6. Alonso-Blas, D.E., Genaim, S.: On the Limits of the Classical Approach to Cost Analysis. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 405–421. Springer, Heidelberg (2012)
7. Anderson, H., Khoo, S.-C., Andrei, Ş., Luca, B.: Calculating Polynomial Runtime Properties. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 230–246. Springer, Heidelberg (2005)
8. Bagnara, R., Mesnard, F., Pescetti, A., Zaffanella, E.: A new look at the automatic synthesis of linear ranking functions. Inf. Comput. 215, 47–67 (2012)
9. Benzinger, R.: Automated higher-order complexity analysis. Theor. Comput. Sci. 318(1-2), 79–103 (2004)
10. Danner, N., Paykin, J., Royer, J.S.: A Static Cost Analysis for a Higher-Order Language. In: PLPV, pp. 25–34. ACM (2013)
11. Debray, S.K., Lin, N.: Cost Analysis of Logic Programs. ACM Trans. Program. Lang. Syst. 15(5), 826–875 (1993)
12. Grobauer, B.: Cost Recurrences for DML Programs. In: ICFP, pp. 253–264. ACM (2001)
13. Gulwani, S., Mehra, J.K., Chilimbi, T.M.: SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In: POPL, pp. 127–139. ACM (2009)
14. Gulwani, S., Zuleger, F.: The Reachability-Bound Problem. In: PLDI, pp. 292–304. ACM (2010)
15. Hoffmann, J., Aehlig, a.K., Hofmannn, M.: Multivariate Amortized Resource Analysis. ACM Trans. Program. Lang. Syst. 34(3), 14:1–14:62 (2012)
16. Simões, H.R., Vasconcelos, P.B., Florido, M., Jost, S., Hammond, K.: Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In: ICFP, pp. 165–176. ACM (2012)
17. Vasconcelos, P.B., Hammond, K.: Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In: Trinder, P., Michaelson, G.J., Peña, R. (eds.) IFL 2003. LNCS, vol. 3145, pp. 86–101. Springer, Heidelberg (2004)
18. Wegbreit, B.: Mechanical Program Analysis. Commun. ACM 18(9), 528–539 (1975)
19. Zuleger, F., Gulwani, S., Sinn, M., Veith, H.: Bound Analysis of Imperative Programs with the Size-Change Abstraction. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 280–297. Springer, Heidelberg (2011)

A The Visit-Bound Method

Although Tree-Sum and Level-Sum can handle a wide range of atomic *CRs*, there are some common patterns that cannot be handled by these approaches.

Example 9. Consider *CR qs* in Fig. 1, and change the expression $\mathsf{nat}(t{-}f)^2$ by $\mathsf{nat}(r)^2$. Then, qs_1 (in Fig. 2) would include $\mathsf{nat}(r)^2$ instead of $\mathsf{nat}(t{-}f)^2$, and qs_{11} (of Ex. 4) would include $\mathsf{nat}(r)^2$ instead of $\mathsf{nat}(t{-}f)^2$. The Tree-Sum and the Level-Sum methods fail on qs_{11}, since the instances of $\mathsf{nat}(r)$, which have the same primary-depth, cannot be bounded by a linear expression. □

This behaviour is common when the cost expression b does not change its value along the recursive calls. In such case, the best we could do is to infer an *UB* on the number of visits to the main equation, and then multiply it by $\hat{b}(\bar{x})$. Note that [3], in such case, would approximate the number of visits by the total number of nodes in the evaluation trees. We aim at a better approximation, by ignoring those nodes that contribute 0 as follows: (1) we construct a *CR* E from D by replacing b by 1; (2) we infer an *UB* for E, using the Tree-Sum method.

Lemma 7. *Let* $E^+(\bar{x})$ *be an* UB *for* E*, then* $E^+(\bar{x}){\cdot}\hat{b}(\bar{x})$ *is an* UB *for* D*.*

Example 10. Let us consider the modified version of qs_{11} (as in Ex. 9). Replacing $b = \mathsf{nat}(r)$ by 1, to generate E as described above, and then solving it using the Tree-Sum method results on the Visit-Bound $\mathsf{nat}(t{-}f{+}2)$. Then, since $\hat{b}(\bar{x})$ in this case is $\mathsf{nat}(r)$, we obtain $\mathsf{nat}(t{-}f){\cdot}\mathsf{nat}(r)$ as an *UB* for qs_{11}. The approach of [3] would obtain $2^{\mathsf{nat}(t-f+1)}{\cdot}\mathsf{nat}(r)$ instead. □

B Solving the universally quantified formulas

In this section we describe how we solve the universally quantified formulas of Lemma 2, Lemma 5, and Lemma 6. Namely, starting from a template linear function $\alpha(\bar{x}) = q_0{+}q_1{\cdot}x_1{+}\ldots{+}q_n{\cdot}x_n$, we find rational values for $q_0, \ldots, q_n$ for which the corresponding formula is valid. Note that our formulas are conjunctions of universally quantified formulas of the following form:

$$\forall \bar{w} : \ \varphi \rightarrow \mathsf{nat}(l_0) \geq q + \mathsf{nat}(l_1) + \ldots + \mathsf{nat}(l_n) \tag{1}$$

where φ defines a closed polyhedron, $q \in \{0, 1\}$, and each l_i is either a linear function over $\bar{w}$, or a template function $\alpha(\bar{x}) = q_0{+}q_1{\cdot}x_1{+}\cdots{+}q_n{\cdot}x_n$ such that $\bar{x} \subseteq \bar{w}$ and $q_i \notin \bar{w}$ (i.e., each q_i is existentially quantified). Our goal is to solve these formulas using linear programming (LP) techniques.

Consider a formula as in (1), but without the nat-expressions, i.e., of the form $\forall \bar{w} : \varphi \rightarrow l_0 \geq q{+}l_1{+}\ldots{+}l_n$. It is known that there is a complete algorithm, based on the use of Farkas' Lemma, for solving a conjunction of such formulas. It has been extensively used for synthesising linear ranking functions [8]. Its complexity is polynomial when the variables $\bar{w}$ range over $\mathbb{Q}$, and exponential when they range over $\mathbb{Z}$. For efficiency considerations, when $\bar{w}$ range over the

integers, as in our case, it is common to relax the formula by assuming that they range over $\mathbb{Q}$. This is sound, since $\mathbb{Z} \subset \mathbb{Q}$, but of course not complete.

Our aim is to transform formulas as (1) to a nat-free as above, and then solve them using the LP based algorithm. Recall that $\mathsf{nat}(l_i) = \max\{l_i, 0\}$. This means that $\mathsf{nat}(l_i)$ can be eliminated by explicitly considering the cases for $l_i \geq 0$ and $l_i \leq 0$ (we use $l_i \geq 0$ and not $l_i > 0$ since in LP constraints must be non-strict). For example, eliminating $\mathsf{nat}(l_0)$ can be done by rewriting (1) as

$$\begin{array}{ll} \forall \bar{w}: \ \varphi \wedge l_0 \geq 0 \rightarrow l_0 \geq q + \mathsf{nat}(l_1) + \ldots + \mathsf{nat}(l_n) & \bigwedge \\ \forall \bar{w}: \ \varphi \wedge l_0 \leq 0 \rightarrow 0 \ \geq q + \mathsf{nat}(l_1) + \ldots + \mathsf{nat}(l_n) & \end{array}$$

In the first equation we assume that l_0 is non-negative, by adding $l_0 \geq 0$ to φ, and thus, on the right-hand side, $\mathsf{nat}(l_0)$ can be replaced by l_0. In the second we assume it is non-positive, by adding $l_0 \leq 0$ to φ, and thus $\mathsf{nat}(l_0)$ can be replaced by 0. This process can be applied iteratively to eliminate each $\mathsf{nat}(l_i)$. There is still one problem in this transformation that prevents us from directly applying the LP techniques: when l_i is a template function, the constraints $l_0 \geq 0$ and $l_0 \leq 0$ are not linear and thus the LP based algorithm cannot be applied. Next we explain how to overcome this problem.

Assume that eliminating the nat-expression results in a formula ξ, and let ξ' be the formula obtained from ξ by simply removing all non-linear constraints. Since all non-linear constraints in ξ appear in the left-hand sides of the implications, we observe that $\xi' \rightarrow \xi$. This, in turn, means that we can solve ξ', using the LP based algorithm, instead of ξ. Although we sacrify completeness, this approach performs well in practice as demonstrated by our experiments. Note that, in addition, we relax ξ' and assume that the universally quantified variables range over $\mathbb{Q}$ (to use the polynomial time algorithm).

Example 11. Consider the atomic *CR* qs_{21} of Ex. 3, which corresponds to the sparse *CR* qs_2 in Fig. 2, and the linear template $h(f,t,r) = h_0{+}h_1{\cdot}f{+}h_2{\cdot}t{+}h_3{\cdot}r$. Applying Lemma 5 to this *CR* and this template, we get the formula $\forall \bar{z}_2 : \Phi_{11} \wedge \Phi_{12}$, where $\bar{z}_2 \equiv \{f, t, r, m', m''\}$ and

$$\begin{array}{l} \Phi_{11} \equiv \psi_2 \rightarrow \mathsf{nat}(h_0{+}h_1{\cdot}f{+}h_2{\cdot}t{+}h_3{\cdot}r) \geq 1{+}\mathsf{nat}(h_0{+}h_1{\cdot}f{+}h_2{\cdot}m'{+}h_3{\cdot}r) \\ \Phi_{12} \equiv \psi_2 \rightarrow \mathsf{nat}(h_0{+}h_1{\cdot}f{+}h_2{\cdot}t{+}h_3{\cdot}r) \geq 1{+}\mathsf{nat}(h_0{+}h_1{\cdot}m''{+}h_2{\cdot}t{+}h_3{\cdot}r) \end{array}$$

Both formulas are the application of Φ_1 of Lemma 5. Removing the nat-expressions in the formula generates the equivalent formula $\forall \bar{z}_2 : \Omega_1 \wedge \Omega_2 \wedge \Omega_3 \wedge \Omega_4$, where

$$\begin{array}{l} \Omega_1 \equiv \mu \wedge h_0{+}h_1{\cdot}f{+}h_2{\cdot}m'{+}h_3{\cdot}r{>}0 \rightarrow \ell \geq 1{+}h_0{+}h_1{\cdot}f{+}h_2{\cdot}m'{+}h_3{\cdot}r \\ \Omega_2 \equiv \mu \wedge h_0{+}h_1{\cdot}f{+}h_2{\cdot}m'{+}h_3{\cdot}r{<}0 \rightarrow \ell \geq 1 \\ \Omega_3 \equiv \mu \wedge h_0{+}h_1{\cdot}m''{+}h_2{\cdot}t{+}h_3{\cdot}r{>}0 \rightarrow \ell \geq 1{+}h_0{+}h_1{\cdot}m''{+}h_2{\cdot}t{+}h_3{\cdot}r \\ \Omega_4 \equiv \mu \wedge h_0{+}h_1{\cdot}m''{+}h_2{\cdot}t{+}h_3{\cdot}r{>}0 \rightarrow \ell \geq 1 \end{array}$$

and $\ell{\equiv}h_0{+}h_1{\cdot}f{+}h_2{\cdot}t{+}h_3{\cdot}r$, $\mu \equiv \psi_2{\wedge}\ell > 0$. Note that the cases $\ell \leq 0$ corresponds to unsatisfiable formulas and are thus ignored.

Now, we remove from this formula the constraints coming from nat expressions in the left-hand side. We then obtain a stronger formula $\forall \bar{z}_2 : \Sigma_1 \wedge \Sigma_2 \wedge \Sigma_3 \wedge \Sigma_4$, where

$\Sigma_1 \equiv \Psi_2 \to \ell \geq 1{+}h_0{+}h_1{\cdot}f{+}h_2{\cdot}m'{+}h_3{\cdot}r$
$\Sigma_2 \equiv \Psi_2 \to \ell \geq 1$
$\Sigma_3 \equiv \Psi_2 \to \ell \geq 1{+}h_0{+}h_1{\cdot}m''{+}h_2{\cdot}t{+}h_3{\cdot}r$

and Σ_4 is exactly like Σ_2. Note that this transformation preserves correctness in the sense that the transformed formula implies the original one, but no vice versa. The value for coefficients $h_0, \ldots, h_3$ is computed using the Farkas Lemma, which infers the following set of constraints $\{h_2{+}h_3{\geq}0, h_1{+}h_2{\geq}0, h_1{\leq}{-}1, h_0{\geq}1\}$, for which one possible solution is $h_0{=}h_2{=}1$ and $h_1{=}{-}1$ what corresponds to the *UB* $\mathsf{nat}(t{-}f{+}1)$ of Ex. 7.

C Proofs

In this Appendix we include the proofs for Obs. 1, Lemma 1, Lemma 2, Lemma 3, Lemma 4,Lemma 5, Lemma 6, and Lemma 7. For convenience, we repeat the text of each statement before the proof.

The semantics of cost relations is formalised in Sec. 2, using evaluation trees (Def. 3). For our proofs, we consider extended evaluation trees of the form $\mathit{node}(C(\bar{v}), \mathcal{E}, \sigma, r, \langle T_1, \ldots, T_k \rangle)$, where $\mathcal{E}$ identifies the cost equation chosen in Def. 3, and σ identifies the assignment chosen in Def. 3.

Proof of Obs. 1

This Observation allows us to decompose a standalone *CR* in several sparse *CRs*, and compute an upper-bound for the standalone cost relation by adding the bounds for the sparse *CRs*. We recall that, in this lemma, $C(\bar{x})$ is a standalone *CR* in which the local cost expression of each equation is in normal form, $P_C = \{P_1, \ldots, P_t\}$ is the set of product cost expressions in the equations of C, and $C_i(\bar{x})$ is the sparse cost relation obtained by removing (setting to zero) every product except for P_i in the equations of C.

Observation 1. If $C_i^+(\bar{x})$ is an *UB* for the sparse *CR* C_i, for all $1 \leq i \leq t$, then $C^+(\bar{x}) = C_1^+(\bar{x}) + \cdots + C_t^+(\bar{x})$ is an *UB* for C.

Proof. The proof of this observation is based on the auxiliary notion of projecting an evaluation tree onto a product cost expression. For any evaluation tree $T{=}\mathit{node}(C(\bar{v}), \mathcal{E}, \sigma, r, \langle T_1, \ldots, T_k \rangle)$ in $\mathit{Trees}(C(\bar{v}))$, the projection of T on the product P_i, written Π_i^T, is defined as:

$$\Pi_i^T = \mathit{node}(C(\bar{v}), \mathcal{E}, \sigma, \sigma(e'), \langle \Pi_i^{T_1}, \ldots, \Pi_i^{T_k} \rangle)$$

where e' is obtained from the cost expression e in the equation $\mathcal{E}$, by setting to zero every product cost expression except for P_i. The projection has two important properties:

1. For each i with $1 \leq i \leq t$, the equations of $C_i(\bar{x})$ have the same constraints as those in $C(\bar{x})$. Hence with the same equation-assignment choices used to build an evaluation tree T for $C(\bar{v})$, we can build one for $C_i(\bar{v})$; in particular, Π_i^T is an evaluation tree for $C_i(\bar{v})$ and $\mathsf{Sum}(\Pi_i^T)$ is a solution of $C_i(\bar{v})$.

2. We can decompose a tree as a sum of the values of its projections. Formally, for every evaluation tree T of $C(\bar{v})$, it holds that $\mathsf{Sum}(T) = \sum_{i=1}^{t} \mathsf{Sum}(\Pi_i^T)$.

Let us consider any $r \in \mathit{Answers}(C(\bar{v}))$. Then $r = \mathsf{Sum}(T)$, for some $T \in \mathit{Trees}(C(\bar{v}))$. By item 2) above, $r = \sum_{i=1}^{t} \mathsf{Sum}(\Pi_i^T)$. Assume that $r_i = \mathsf{Sum}(\Pi_i^T)$. Then, by item 1) $r_i \in \mathit{Answers}(C_i(\bar{v}))$, i.e., $r_i \leq C_i^+(\bar{v})$. Thus $r_1 + \ldots + r_n \leq C_1^+(\bar{v}) + \cdots + C_t^+(\bar{v})$, i.e., $r \leq C^+(\bar{v})$ for any $r \in \mathit{Answers}(C(\bar{v}))$. This means that $C^+(\bar{x})$ is an upper-bound for $C(\bar{x})$.

Proof of Lemma 1

This lemma allows us to solve a sparse cost relation by solving the atomic cost relation that corresponds to one factor, and multiply the obtained bound by the maximisation of the other factors.

Lemma 1. *Let C_i be a* sparse CR*, $b_{i1} \cdot \cdots \cdot b_{ip_i}$ the product in its main equation, and C_{ij} an atomic* CR *of C_i. If $C_{ij}^+(\bar{x})$ is an* UB *for $C_{ij}(\bar{x})$, then $C_i^+(\bar{x}) = C_{ij}^+(\bar{x}) \cdot \prod_{k \neq j} \hat{b}_{ik}(\bar{x})$ is an* UB *for C_i.*

Proof. Similarly as done in the proof of Obs. 1, let us first define the projection of an evaluation tree $T = \mathit{node}(C_i(\bar{v}), \mathcal{E}, \sigma, r, \langle T_1, \ldots T_k \rangle)$ for $C_i(\bar{v})$ on the product b_{ij}, written Π_{ij}^T, as:

$$\Pi_{ij}^T = \mathit{node}(C_i(\bar{v}), \mathcal{E}, \sigma, \sigma(e'), \langle \Pi_{ij}^{T_1}, \ldots, \Pi_{ij}^{T_k} \rangle)$$

where e' is obtained from the expression e of the cost equation $\mathcal{E}$, by setting to 1 every product except for b_{ij}, if present. As reasoned in Obs. 1, it can be noted that for every evaluation tree T of $C_i(\bar{v})$ and for any basic cost expression b_{ij}, the projection Π_{ij}^T is an evaluation tree for $C_{ij}(\bar{v})$. We prove now by induction on the depth (maximum length from the root to a leaf) of T that:

"For any $T \in \mathit{Trees}(C_i(\bar{v}))$, it holds that $\mathsf{Sum}(T) \leq \mathsf{Sum}(\Pi_{ij}^T) \cdot \prod_{k \neq j} \hat{b}_{ik}(\bar{v})$"

For readability, in this proof $\hat{\mathbf{P}}(\bar{v})$ to denote $\prod_{k \neq j} \hat{b}_{ik}(\bar{v})$. Note that since each $\hat{b}_{ik}(\bar{x})$ is a maximisation of b_{ik}, then $\hat{\mathbf{P}}(\bar{x})$ is a maximisation of $\prod_{k \neq j} b_{ik}$.

Base case (depth=0). The value $\mathsf{Sum}(T)$ of a leaf of an evaluation tree of the form $T{=}\mathit{node}(C(\bar{v}), \mathcal{E}, \sigma, r, \langle\rangle)$ is just r. We distinguish two cases:

1. If $r = 0$, i.e. the leaf is not built using the main equation. Then $\mathsf{Sum}(T) = \mathsf{Sum}(\Pi_{ij}^T) = 0$ and $\mathsf{Sum}(T) \leq \mathsf{Sum}(\Pi_{ij}^T) \cdot \hat{\mathbf{P}}(\bar{v})$ trivially holds.
2. If $r \neq 0$ then $r{=}\sigma(b_{ij} \cdot \prod_{k \neq j} b_{ik})$. Since $\hat{\mathbf{P}}$ is a maximisation of $\prod_{k \neq j} b_{ik}$ we get that $\sigma(b_{ij} \cdot \prod_{k \neq j} b_{ik}) {\leq} \sigma(b_{ij} \cdot \hat{\mathbf{P}})$. But it holds that $\mathit{vars}(\hat{\mathbf{P}}) \subseteq \bar{x}$, i.e., $\hat{\mathbf{P}}$ is ground. Hence $\sigma(b_{ij} \cdot \hat{\mathbf{P}}) = \sigma(b_{ij}) \cdot \hat{\mathbf{P}}$. Then $\mathsf{Sum}(T) = \sigma(b_{ij}) \cdot \sigma(\prod_{k \neq j} b_{ik}) \leq \sigma(b_{ij}) \cdot \hat{\mathbf{P}}(\bar{v}) = \mathsf{Sum}(\Pi_{ij}^T) \cdot \hat{\mathbf{P}}$.

Inductive case (depth $>$ 0). Let us consider then a non-leaf evaluation tree $node(C(\bar{v}), \mathcal{E}, \sigma, r, \langle T_1, \ldots, T_k \rangle)$, where $\mathcal{E} = \langle C(\bar{x}) = e + \sum_{j=1}^{k} C(\bar{y}_j), \varphi \rangle$. By induction hypothesis it hods that $\mathsf{Sum}(T_p) \leq \mathsf{Sum}(\Pi_{ij}^{T_p}) \cdot \hat{\mathbf{P}}(\sigma(\bar{y}_p))$, for all $1 \leq p \leq k$. We distinguish two cases:

1. If $r = 0$, that is if T is not built from the main equation. Then:

$$\begin{array}{lll} \mathsf{Sum}(T) = \sum_{p=1}^{k} \mathsf{Sum}(T_p) & \leq & (1) \\ = \mathsf{Sum}(\Pi_{ij}^{T_1}) \cdot \hat{\mathbf{P}}(\sigma(\bar{y}_1)) + \cdots + \mathsf{Sum}(\Pi_{ij}^{T_k}) \cdot \hat{\mathbf{P}}(\sigma(\bar{y}_k)) & \leq & (2) \\ (\sum_{p=1}^{k} \mathsf{Sum}(\Pi_{ij}^{T_p})) \cdot \hat{\mathbf{P}}(\bar{v}) & = & \\ \mathsf{Sum}(\Pi_{ij}^{T}) \cdot \hat{\mathbf{P}}(\bar{v}) & & \end{array}$$

   where (1)="Induction Hypothesis" and (2)="definition of maximisation"
2. If $r \neq 0$, then $\mathsf{Sum}(T) = r + \sum_{p=1}^{k} \mathsf{Sum}(T_p)$ and $r = \sigma(b_{ij} \cdot \prod_{k \neq j} b_{ik})$. Reasoning similarly to point 2) of the base case, we get $\sigma(b_{ij} \cdot \prod_{k \neq j} b_{ik}) \leq \sigma(b_{ij} \cdot \hat{\mathbf{P}}) = \sigma(b_{ij}) \cdot \hat{\mathbf{P}}$, i.e., $r \leq \sigma(b_{ij}) \cdot \hat{\mathbf{P}}$. Therefore, $\mathsf{Sum}(T) \leq (\sigma(b_{ij}) + \sum_{p=1}^{k} \mathsf{Sum}(\Pi_{ij}^{T_p})) \cdot \hat{\mathbf{P}}$. But $\mathsf{Sum}(\Pi_{ij}^{T}) = \sigma(b_{ij}) + \sum_{p=1}^{k} \mathsf{Sum}(\Pi_{ij}^{T_p})$, then we get $\mathsf{Sum}(T) \leq \mathsf{Sum}(\Pi_{ij}^{T}) \cdot \hat{\mathbf{P}}$.

Now, for every evaluation tree T of $C_i(\bar{v})$, it holds that $\mathsf{Sum}(T) \leq \mathsf{Sum}(\Pi_{ij}^{T}) \cdot \hat{\mathbf{P}}$, and Π_{ij}^{T} is an evaluation tree of $C_{ij}(\bar{v})$. The lemma assumes that $C_{ij}^{+}(\bar{x})$ is an upper-bound of $C_{ij}(\bar{x})$, so $\mathsf{Sum}(\Pi_{ij}^{T}) \leq C_{ij}^{+}(\bar{v})$. So for every evaluation tree T of $C_i(\bar{v})$ we have that $\mathsf{Sum}(T) \leq C_{ij}^{+}(\bar{v}) \cdot \hat{\mathbf{P}}$, which means that $C_{ij}^{+}(\bar{x}) \cdot \hat{\mathbf{P}}$ is an upper-bound of $C_i(\bar{x})$.

Proof of Lemma 2

This lemma describes the Tree-Sum method for solving an atomic cost relation $D(\bar{x})$ that, in its main equation, has $\mathsf{nat}(l)$ as basic cost expression.

Lemma 2. *Let $\alpha(\bar{x}) = q_0 + q_1 \cdot x_1 + \cdots + q_n \cdot x_n$, and define:*

$$\Psi_1 \triangleq \forall \bar{w}_1 : \varphi_1 \rightarrow \mathsf{nat}(\alpha(\bar{x})) \geq \mathsf{nat}(l) + \sum_{j=1}^{k_1} \mathsf{nat}(\alpha(\bar{y}_{1j}))$$
$$\Psi_2 \triangleq \bigwedge_{i=2}^{m} \forall \bar{w}_i : \varphi_i \rightarrow \mathsf{nat}(\alpha(\bar{x})) \geq \sum_{j=1}^{k_i} \mathsf{nat}(\alpha(\bar{y}_{ij}))$$

If $\Psi_1 \wedge \Psi_2$ is valid, then $\mathsf{nat}(\alpha(\bar{x}))$ is an UB *for the atomic* CR *D.*

Proof. We prove by induction on the depth of a tree T that, under the conditions of the lemma, it holds that:

"$\mathsf{nat}(\alpha(\bar{v})) \geq \mathsf{Sum}(T)$ *for every evaluation tree T of $D(\bar{v})$*"

Base case (depth=0). In a leaf tree $T = node(D(\bar{v}), \mathcal{E}, \sigma, r, \langle\rangle)$, the value of $\mathsf{Sum}(T)$ is just r. If $r = 0$, i.e if T is not built from the main equation, then the result trivially holds. Otherwise, if $r \neq 0$, i.e,, the main equation has been

applied, then $r{=}\sigma(\mathsf{nat}(l))$. But by definition of evaluation tree, it holds $\sigma \models \varphi_1$. Thus by condition Ψ_1 it holds that $\mathsf{nat}(\alpha(\bar{v})){\geq}\sigma(\mathsf{nat}(l)) = r = \mathsf{Sum}(T)$.

Inductive case (depth $\geq$ 0). Any evaluation tree generated from a cost equation $\mathcal{E}_p = \langle D(\bar{x}) = \mathsf{nat}(l){+}\sum_{j=1}^{k_p} D(\bar{y}_{pj}), \varphi_p\rangle$ has the form $T = \mathit{node}(D(\bar{v}), \mathcal{E}_p, \sigma, r, T_{p1}, \ldots, T_{pk_p})$, and it verifies that $\mathsf{Sum}(T) = r + \sum_{j=1}^{k_p} \mathsf{Sum}(T_{pj})$. By induction hypothesis it holds that $\sigma(\mathsf{nat}(\alpha(\bar{y}_{pj}))) \geq \mathsf{Sum}(T_{pj})$, $1 \leq j \leq k_p$. Then we get that $\sum_{j=1}^{k_p} \sigma(\mathsf{nat}(\alpha(\bar{y}_{pj}))) \geq \sum_{j=1}^{k_p} \mathsf{Sum}(T_{pj})$. We distinguish two cases, depending on if $p = 1$, i.e., if we are in presence of the main equation or not.

1. If $p \neq 1$, that is if T is not built from the main equation, then $r{=}0$ and $\mathsf{Sum}(T){=}\sum_{j=1}^{k_p} \mathsf{Sum}(T_{pj})$. Since $\sigma \models \varphi_p$, the condition Ψ_2 allows to ensure that $\mathsf{nat}(\alpha(\bar{v})){\geq}\sum_{j=1}^{k_p} \sigma(\mathsf{nat}(\alpha(\bar{y}_{ij})))$, which in turn is greater than $\mathsf{Sum}(T)$.
2. If $p = 1$, i.e., the evaluation tree T is built from the main equation, then $r = \sigma(\mathsf{nat}(l))$. Since $\sigma \models \varphi_1$, condition Ψ_1 implies that $\sigma(\mathsf{nat}(\alpha(\bar{v}))) \geq \sigma(\mathsf{nat}(l)){+}\sum_{j=1}^{k_1} \sigma(\mathsf{nat}(\alpha(\bar{y}_{ij})))$. Since $\sum_{j=1}^{k_1} \sigma(\mathsf{nat}(\alpha(\bar{y}_{ij}))) \geq \sum_{j=1}^{k_1} \mathsf{Sum}(T_{1j})$, we get that $\mathsf{nat}(\alpha(\bar{v})) \geq \mathsf{Sum}(T)$.

Proof of Lemma 3

This lemma extends the Tree-Sum method to compute an upper-bound for any kind of atomic cost relation. We assume that $\mathsf{nat}(l)$ is the nat-expression involved in the basic cost expression of the main equation. We recall that, in this lemma, $D(\bar{x})$ is an atomic cost relation that in its main equation has as local expression either $\mathsf{nat}(l)$, or $a^{\mathsf{nat}(l)}$, or $\mathit{log}_a(1 + \mathsf{nat}(l))$, or a constant q. Also, $E(\bar{x})$ is an atomic cost relation obtained from D by replacing its main expression with the $\mathsf{nat}(l)$ subexpression, or with $\mathsf{nat}(q)$ if it is a constant. Note that in the lemma below we have used the constant $\mathit{log}_a\mathsf{e}$ instead of 1.5 since $\mathit{log}_a\mathsf{e}$ is the most precise bound.

Lemma 3. *Let* $\mathsf{nat}(\alpha(\bar{x}))$ *be an* UB *for* E*, and let*

$$D^+(\bar{x}) = \begin{cases} \mathsf{nat}(\alpha(\bar{x})) & b = q \vee b = \mathsf{nat}(l) \\ \mathsf{nat}(\alpha(\bar{x}))\cdot \mathit{log}_a\mathsf{e} & b = \log_a(1 + \mathsf{nat}(l)) \\ a^{\mathsf{nat}(\alpha(\bar{x}))} - 1 & b = a^{\mathsf{nat}(l)} - 1 \end{cases}$$

Then, $D^+(\bar{x})$ *is an* UB *for* D.

Proof. For the case in which the basic cost expression in $D(\bar{x})$ has the form either $\mathsf{nat}(l)$ or $q \in \mathbb{Q}^+$, we can apply directly Lemma 2. For the rest of cases we reason as follows:

1. Suppose that the basic cost expression in $D(\bar{x})$ has the form $a^{\mathsf{nat}(l)} - 1$. Considering that $\mathsf{nat}(\alpha(\bar{x}))$ is an *UB* for $E(\bar{x})$, then for any $\bar{v}$, and evaluation tree for $E(\bar{v})$ it holds that:

$$(*)\ \mathsf{nat}(\alpha(v)) \geq \sigma_1(\mathsf{nat}(l)) + \ldots + \sigma_h(\mathsf{nat}(l))$$

But for any $r_1, r_2 \in \mathbb{R}^+$, $a \in \mathbb{N}$, $a \geq 2$, it holds that $a^{r_1+r_2} - 1 \geq a^{r_1} - 1 + a^{r_2} - 1$. Applying this result we get $a^{\sigma_1(\mathsf{nat}(l))+\ldots+\sigma_h(\mathsf{nat}(l))} \geq a^{\sigma_1(\mathsf{nat}(l))} - 1 + \ldots + a^{\sigma_h(\mathsf{nat}(l))} - 1$. Then from $(*)$, it holds $a^{\mathsf{nat}(\alpha(\bar{v}))} - 1 \geq a^{\sigma_1(\mathsf{nat}(l))} - 1 + \ldots + a^{\sigma_h(\mathsf{nat}(l))} - 1$, i.e., $a^{\mathsf{nat}(\alpha(\bar{x}))} - 1$ is an *UB* for D.

2. Now, assume that the basic cost expression in $D(\bar{x})$ is $\log_a(1 + \mathsf{nat}(l))$. In this case we use the following property of logarithms:

$$(**)\ \log_a P = \log_{\mathsf{e}} P \cdot \log_a \mathsf{e}$$

Now, given any $\bar{v}$ and evaluation tree for $D(\bar{v})$, it holds:

$$\begin{array}{lll} \mathsf{Sum}(T) = \sigma_1(\log_a(1+\mathsf{nat}(l))) + \ldots \sigma_h(\log_a(1+\mathsf{nat}(l)))) & = & \text{(by } (**)\text{)} \\ \log_a \mathsf{e} \cdot (\sigma_1(\log_{\mathsf{e}}(1+\mathsf{nat}(l))) + \ldots + \sigma_h(\log_{\mathsf{e}}(1+\mathsf{nat}(l)))) & \leq & \\ \log_a \mathsf{e} \cdot (\sigma_1(\mathsf{nat}(l)) + \ldots + \sigma_h(\mathsf{nat}(l))) & \leq & \text{(by } (*)\text{)} \\ \log_a \mathsf{e} \cdot \mathsf{nat}(\alpha(\bar{v})) & & \end{array}$$

i.e., $\log_a \mathsf{e} \cdot \mathsf{nat}(\alpha(\bar{x}))$ is an *UB* for D.

The Level-Sum method obtains an upper-bound for a cost relation as the product of two expressions, each one an upper-bound of a different measure of the evaluation tree. The first measure is the number of levels of the main equation, called the primary depth of the tree. The second measure is an upper-bound on the total cost of each level. Although these notions are explained in Sec. 3.2, we give here a formal definition.

Let $C(\bar{x})$ be a sparse cost relation. We define formally the primary depth $pdepth(node(C(\bar{v}), \mathcal{E}, \_, \_, \langle T_1, \ldots, T_k \rangle))$ of an evaluation tree T of $C(\bar{v})$ as:

$$\max\{pdepth(T_j) \mid 1 \leq j \leq k\} + \begin{cases} 1 & \mathcal{E} \text{ is the main equation} \\ 0 & \text{otherwise} \end{cases}$$

In the following, when needed, we denote by $\mathcal{E}^M$ to the main equation of a *CR*. Now we define formally the Level-Sum of the i-th level of T, written $\mathsf{SumLevel}(T, i)$, as:

$$\begin{array}{ll} \mathsf{SumLevel}(node(C(\bar{v}), \mathcal{E}^M, \_, r, \langle T_1, \ldots, T_k \rangle), 1) = r & \\ \mathsf{SumLevel}(node(C(\bar{v}), \mathcal{E}^M, \_, r, \langle T_1, \ldots, T_k \rangle), i) = \sum_{j=1}^{k} \mathsf{SumLevel}(T_j, i-1) & i \geq 2 \\ \mathsf{SumLevel}(node(C(\bar{v}), \mathcal{E}, \_, 0, \langle T_1, \ldots, T_k \rangle), i) = \sum_{j=1}^{k} \mathsf{SumLevel}(T_j, i) & \end{array}$$

and we finally define $\mathsf{SumLevel}^*(T)$ as $\max\{\mathsf{SumLevel}(T, i) \mid i \geq 1\}$.

We can now prove Lemma 4, which is the basis for the Level-Sum method to solve an atomic cost relation.

Lemma 4. *Let $\alpha_1(\bar{x})$ and $\alpha_2(\bar{x})$ be two linear functions such that* $\mathsf{nat}(\alpha_1(\bar{x}))$ *and* $\mathsf{nat}(\alpha_2(\bar{x}))$ *are respectively* UBs *on the primary-depth and Level-Sum of D. Then,* $\mathsf{nat}(\alpha_1(\bar{x})) \cdot \mathsf{nat}(\alpha_2(\bar{x}))$ *is an* UB *for D.*

Proof. Let $T = node(D(\bar{v}), \mathcal{E}, \_, r, \langle T_1, \ldots, T_k \rangle)$ be any evaluation tree of $D(\bar{v})$, for any input $\bar{v}$. It trivially holds that:

$$\mathsf{Sum}(T) = \sum_{i=1}^{k} \mathsf{SumLevel}(T,i) = \sum_{i=1}^{pdepth(T)} \mathsf{SumLevel}(T,i)$$

Since $\mathsf{SumLevel}(T,i) \leq \mathsf{SumLevel}^*(T)$ for each $1 \leq i \leq k$, we get that $\mathsf{Sum}(T) \leq pdepth(T) \cdot \mathsf{SumLevel}^*(T)$. By hypothesis it holds that $\mathsf{nat}(\alpha_1(\bar{v})) \geq pdepth(T)$ and $\mathsf{nat}(\alpha_2(\bar{v})) \geq \mathsf{SumLevel}^*(T)$, then we have that $\mathsf{Sum}(T) \leq \mathsf{nat}(\alpha_1(\bar{v})) \cdot \mathsf{nat}(\alpha_2(\bar{v}))$, for any tree $T \in Trees(D(\bar{v}))$. Hence $\mathsf{nat}(\alpha_1(\bar{x})) \cdot \mathsf{nat}(\alpha_2(\bar{x}))$ is an *UB* for $D(\bar{x})$.

Proof of Lemma 5

This lemma provides an inductive sufficient condition for an upper-bound on the primary depth of a cost relation.

Lemma 5. *Let $\alpha(\bar{x}) = q_0 + q_1 \cdot x_1 + \cdots + q_n \cdot x_n$, and define:*

$$\Phi_1 \triangleq \begin{cases} \forall \bar{w}_1 : \varphi_1 \rightarrow \mathsf{nat}(\alpha(\bar{x})) \geq 1 & k = 0 \\ \bigwedge_{j=1}^{k_1} \forall \bar{w}_1 : \varphi_1 \rightarrow \mathsf{nat}(\alpha(\bar{x})) \geq 1 + \mathsf{nat}(\alpha(\bar{y}_{1j})) & k \geq 1 \end{cases}$$

$$\Phi_2 \triangleq \bigwedge_{i=2}^{m} \bigwedge_{j=1}^{k_i} \forall \bar{w}_i : \varphi_i \rightarrow \mathsf{nat}(\alpha(\bar{x})) \geq \mathsf{nat}(\alpha(\bar{y}_{ij}))$$

If $\Phi_1 \wedge \Phi_2$ is valid, then $\mathsf{nat}(\alpha(\bar{x}))$ is an UB *on the primary-depth of D.*

Proof. The statement that $\mathsf{nat}(\alpha(\bar{x}))$ is an upper-bound on the primary depth of $D(\bar{x})$ can be reformulated as:

"For every evaluation tree T of $D(\bar{v})$, it holds that $pdepth(T) \leq \mathsf{nat}(\alpha(\bar{v}))$"

Let us prove this last result by induction on the depth of the evaluation tree.

Base case (depth=0). For a leaf tree $T = node(D(\bar{v}), \mathcal{E}, \sigma, \_, \langle\rangle)$, we distinguish two cases:

- If $\mathcal{E} \neq \mathcal{E}^M$, then T is not a primary node and $pdepth(T)$=0. Then $\mathsf{nat}(\alpha(\bar{v})) \geq 0$ holds by the definition of nat.
- If $\mathcal{E} = \mathcal{E}^M$, then T is a primary node and $pdepth(T)$=1. Since σ satisfies φ_1, then the condition Φ_1 implies that $\mathsf{nat}(\alpha(\bar{v})) \geq 1$.

Inductive case (depth $\geq$ 0). Then $T = node(D(\bar{v}), \mathcal{E}_p, \sigma, \_, \langle T_{p1}, \ldots, T_{pk_p} \rangle)$, where $\mathcal{E}_p = \langle e_p + \sum_{j=1}^{k_p} D(\bar{y}_{pj}), \varphi_p \rangle$ and each T_{pj} is an evaluation tree for the call $D(\sigma(\bar{y}_{pj}))$. By induction hypothesis it holds that $\sigma(\mathsf{nat}(\alpha(\bar{y}_{pj}))) \geq pdepth(T_{pj})$ holds for each $1 \leq j \leq k_p$. Then:

$$\max\{\sigma(\mathsf{nat}(\alpha(\bar{y}_{pj}))) \mid 1 \leq j \leq k_p\} \geq \max\{pdepth(T_{pj}) \mid 1 \leq j \leq k_p\}$$

We consider two cases depending on $\mathcal{E}$:

– If $\mathcal{E} \neq \mathcal{E}^M$, i.e., the root is not a primary node, then it holds that $pdepth(T) = \max\{pdepth(T_{pj}) \mid 1 \leq j \leq k_p\}$. Since σ satisfies φ_p, the condition Φ_2 implies that $\bigwedge_{j=1}^{k_p} \mathsf{nat}(\alpha(\bar{v})) \geq \sigma(\mathsf{nat}(\alpha(\bar{y}_{pj})))$, which implies $\mathsf{nat}(\alpha(\bar{v})) \geq \max\{\sigma(\mathsf{nat}(\alpha(\bar{y}_{pj}))) \mid 1 \leq j \leq k_p\}$. This, in turn, means that $\mathsf{nat}(\alpha(\bar{v})) \geq pdepth(T)$.
– If $\mathcal{E} = \mathcal{E}^M$, i.e., we have applied the main equation, then the root of the tree is a primary node, $p = 1$ and $pdepth(T)=1 + \max\{pdepth(T_{1j}) \mid 1 \leq j \leq k_1\}$. The assignment σ must satisfy φ_1, which by condition Φ_1. This implies that $\bigwedge_{j=1}^{k_1} \mathsf{nat}(\alpha(\bar{v}))\geq 1+\sigma(\mathsf{nat}(\alpha(\bar{y}_{1j})))$, which is a sufficient condition of $\mathsf{nat}(\alpha(\bar{v})) \geq 1+ \max\{\sigma(\mathsf{nat}(\alpha(y_{1j}))) \mid 1 \leq j \leq k_1\}$, and therefore that $\mathsf{nat}(\alpha(\bar{v})) \geq pdepth(T)$.

Proof of Lemma 6

This lemma gives a sufficient condition for a function being an upper-bound on the Level-Sum of a cost relation.

Lemma 6. *Let* $\alpha(\bar{x}) = q_0 + q_1 * x_1 + \cdots + q_n * x_n$, *and define:*

$$\begin{array}{ll} \Pi_1 \triangleq \forall \bar{w}_1 : \varphi_1 \rightarrow & \mathsf{nat}(\alpha(\bar{x})) \geq \mathsf{nat}(l) \\ \Pi_2 \triangleq \bigwedge_{i=1}^{m} \forall \bar{w}_i : \varphi_i \rightarrow & \mathsf{nat}(\alpha(\bar{x})) \geq \sum_{j=1}^{k_i} \mathsf{nat}(\alpha(\bar{y}_{ij})) \end{array}$$

If $\Pi_1 \wedge \Pi_2$ *is valid, then* $\mathsf{nat}(\alpha(\bar{x}))$ *is an* UB *on the Level-Sum of* D.

Proof. The statement that $\mathsf{nat}(\alpha(\bar{x}))$ is an upper-bound on the Level-Sum of D can be reformulated as:

"For every evaluation tree T *of* $D(\bar{v})$, *it holds that* $\mathsf{SumLevel}^*(T) \leq \mathsf{nat}(\alpha(\bar{v}))$*"*

Let us proof this statement by induction on the depth of evaluation trees.

Base case (depth=0). In a leaf tree $T=node(D(\bar{v}), \mathcal{E}, \sigma, r, \langle\rangle)$, we distinguish two cases:

1. If T is not primary node, i.e., we have not applied the main equation, then $\mathsf{SumLevel}^*(T) = 0$ and $\mathsf{nat}(\alpha(\bar{v})) \geq 0$ trivially holds.
2. If T is a primary node then $\mathsf{SumLevel}^*(T) = \sigma(\mathsf{nat}(l))$. Since $\sigma \models \varphi_1$, condition Π_1 implies that $\mathsf{nat}(\alpha(\bar{v})) \geq \sigma(\mathsf{nat}(l))$. Then $\mathsf{nat}(\alpha(\bar{v})) \geq \mathsf{SumLevel}^*(T)$.

Inductive case (depth $\geq$ 0). Suppose that the non-leaf tree has been generated by applying an equation $\mathcal{E}_p = \langle D(\bar{x})=\mathsf{nat}(l)+\sum_{j=1}^{k_p} D(\bar{y}_{pj}), \varphi_p\rangle$, then the evaluation tree will have the form $T=node(D(\bar{v}), \mathcal{E}_p, \sigma, r, \langle T_{p1}, \ldots, T_{pk_p}\rangle)$. By induction hypothesis it holds that $\alpha(\sigma(\bar{y}_{pj})) \geq \mathsf{SumLevel}^*(T_{pj})$, for each $1 \leq j \leq k_p$, i.e., $\sum_{j=1}^{k_p}(\sigma(\bar{y}_{pj})) \geq \sum_{j=1}^{k_p} \mathsf{SumLevel}(T_{pj})$. But, from the definition of $\mathsf{SumLevel}^*$ we have that $\mathsf{SumLevel}^*(T_{pj}) \geq \mathsf{SumLevel}(T_{pj}, i)$, for all $i \geq 1$. Then $\sum_{j=1}^{k_p}(\sigma(\bar{y}_{pj})) \geq \sum_{j=1}^{k_p} \mathsf{SumLevel}(T_{pj}, i)$, for all $i \geq 1$. Since $\sigma \models \varphi_p$, condition Π_2 implies that $\mathsf{nat}(\alpha(\bar{v})) \geq \sum_{j=1}^{k_p} \mathsf{nat}(\alpha(\sigma(\bar{y}_{pj})))$. Hence:

(†) $\mathsf{nat}(\alpha(\bar{v})) \geq \sum_{j=1}^{k_p} \mathsf{SumLevel}(T_{pj}, i)$, for all $i \geq 1$

Now, we distinguish two cases depending on the selected cost equation:

1. If the selected cost equation is different from the main one then the root of T is not a primary node and thus the levels of T are those of its subtrees. This means that for all $i \geq 1$, $\sum_{j=1}^{k_p} \mathsf{SumLevel}(T_{pj}, i) = \mathsf{SumLevel}(T, i)$, i.e., $\mathsf{nat}(\alpha(\bar{v})) \geq \sum_{j=1}^{k_p} \mathsf{SumLevel}(T_{pj}, i)$, i.e, that $\mathsf{nat}(\alpha(\bar{v})) \geq \mathsf{SumLevel}^*(T)$.
2. If the root of T is a primary node, i.e., we have applied the main equation ($p = 1$), we have to distinguish the first level, at the root, from the other levels, at the children subtrees.
   (a) With respect to the root level, $\mathsf{SumLevel}(T, 1) = r = \sigma(\mathsf{nat}(l))$, but since $\sigma \models \varphi_1$, the condition Π_1 implies that $\mathsf{nat}(\alpha(\bar{v})) \geq \sigma(\mathsf{nat}(l))$.
   (b) With respect to the levels below the root, condition (†) implies that for each $i \geq 2$, $\mathsf{nat}(\alpha(\bar{v})) \geq \mathsf{SumLevel}(T, i)$.

   So, for each level $i \geq 1$ we have that $\mathsf{nat}(\alpha(\bar{v})) \geq \mathsf{SumLevel}(T, i)$, and therefore $\mathsf{nat}(\alpha(\bar{v})) \geq \mathsf{SumLevel}^*(T)$.

Proof of Lemma 7

We recall from Sec. A that, in this method, we have an atomic cost relation $D(\bar{x})$, and b is the basic cost expression in its main equation. This method gets an upper-bound for an atomic cost relation by simply multiplying the maximisation $\hat{b}(\bar{x})$ of the local cost by an upper-bound on the number of applications of the main equation. To obtain this upper-bound, we construct a *CR* E from D replacing b by 1, and we infer an *UB* for E.

Lemma 7. *Let* $E^+(\bar{x})$ *be an* UB *for* E*, then* $E^+(\bar{x}) \cdot \hat{b}(\bar{x})$ *is an* UB *for* D*.*

Proof. Similarly as we did to prove Obs. 1 and Lemma 1, we first define the projection of an evaluation tree $T = node(D(\bar{v}), \mathcal{E}, \sigma, r, \langle T_1, \ldots T_k \rangle)$ for $D(\bar{v})$ on its node-count, written $\Pi^T_\#$, as:

$$\Pi^T_\# = node(D(\bar{v}), \mathcal{E}, \sigma, \#, \langle \Pi^{T_1}_\#, \ldots, \Pi^{T_k}_\# \rangle)$$

where # is 1 if T is built with the main equation, and # is 0 otherwise. As reasoned in Obs. 1 and Lemma 1, it can be noted that for every evaluation tree T of $D(\bar{v})$, the projection $\Pi^T_\#$ is an evaluation tree for $E(\bar{v})$. We prove now by induction on the depth of T that:

"For any $T \in Trees(D(\bar{v}))$*, it holds that* $\mathsf{Sum}(T) \leq \mathsf{Sum}(\Pi^T_\#) \cdot \hat{b}(\bar{v})$*"*

Base case (depth=0). The value $\mathsf{Sum}(T)$ of a leaf of an evaluation tree of the form $T{=}node(D(\bar{v}), \mathcal{E}, \sigma, r, \langle\rangle)$ is just r. We distinguish two cases:

1. If $\mathcal{E}$ is different from the main equation, i.e. the leaf is not built using the main equation, then $\mathsf{Sum}(T) = \mathsf{Sum}(\Pi^T_\#) = 0$ and $\mathsf{Sum}(T) \leq \mathsf{Sum}(\Pi^T_\#) \cdot \hat{b}(\bar{v})$ trivially holds.

2. If $\mathcal{E}$ is the main equation, then $r{=}\sigma(b)$, and $\mathsf{Sum}(\Pi^T_\#) = 1$. Since $\hat{b}(\bar{v})$ is a maximisation of $\sigma(b)$ we get that $\sigma(b){\leq}\sigma(\hat{b}(\bar{v})) = \hat{b}(\bar{v})$. Then $\mathsf{Sum}(T) = \sigma(b) \leq 1{\cdot}\hat{b}(\bar{v}) = \mathsf{Sum}(\Pi^T_\#){\cdot}\hat{b}(\bar{v})$.

Inductive case (depth > 0). Let us consider a non-leaf evaluation tree T of the form $node(D(\bar{v}), \mathcal{E}_p, \sigma, r, \langle T_{p1}, \ldots, T_{pk_p}\rangle)$. By induction hypothesis it holds that $\mathsf{Sum}(T_{pj}){\leq}\mathsf{Sum}(\Pi^{T_{pj}}_\#){\cdot}\hat{b}(\bar{v}_{pj})$, for all $1 \leq j \leq k_p$. By definition of maximisation it holds that that $\sum_{j=1}^{k_p} \mathsf{Sum}(T_{pj}) \leq \hat{b}(\bar{v}){\cdot}\sum_{j=1}^{k_p} \mathsf{Sum}(\Pi^{T_{pj}}_\#)$. We distinguish two cases:

1. If $\mathcal{E}$ is different from the main equation then:

$$\begin{array}{lll} \mathsf{Sum}(T) = \sum_{j=1}^{k_p} \mathsf{Sum}(T_{pj}) & \leq & \text{(by induction hypothesis)} \\ \hat{b}(\bar{v}) \cdot \sum_{j=1}^{k_p} \mathsf{Sum}(\Pi^{T_{pj}}_\#) = & & (T \text{ is not built from the main equation}) \\ \hat{b}(\bar{v}) \cdot \mathsf{Sum}(\Pi^T_\#) & & \end{array}$$

2. If $\mathcal{E}$ is the main equation, then $\mathsf{Sum}(T){=}r{+}\sum_{j=1}^{k_p} \mathsf{Sum}(T_{pj})$ and $r{=}\sigma(b)$. Like in point 2) of the base case, we get $\sigma(b) \leq \sigma(\hat{b}(\bar{v})) = \hat{b}(\bar{v})$, i.e., $r \leq \hat{b}(\bar{v})$. Therefore, $\mathsf{Sum}(T) \leq \hat{b}(\bar{v}){+}\sum_{j=1}^{k_p} \mathsf{Sum}(T_{pj}) \leq \hat{b}(\bar{v}){\cdot}\left(1 + \sum_{j=1}^{k_p} \mathsf{Sum}(\Pi^{T_{pj}}_\#)\right)$. Since T is built from the main equation, then $\mathsf{Sum}(\Pi^T_\#) = 1{+}\sum_{j=1}^{k_p} \mathsf{Sum}(\Pi^{T_{pj}}_\#)$, so we get $\mathsf{Sum}(T) \leq \mathsf{Sum}(\Pi^T_\#){\cdot}\hat{b}(\bar{v})$.

If we assume that $E^+(\bar{x})$ is an upper bound of $E(\bar{x})$, then $\mathsf{Sum}(\Pi^T_\#) \leq E^+(\bar{v})$, and for every evaluation tree T of $D(\bar{v})$ we have that $\mathsf{Sum}(T) \leq E^+(\bar{v}){\cdot}\hat{b}$, which means that $E^+(\bar{x}){\cdot}\hat{b}$ is an upper-bound of $D(\bar{x})$.

Printed in Madrid, Spain, on 25th March 2014.